Universidad
Carlos III de Madrid

Escuela Politécnica Superior

Ingeniería de Telecomunicación

Departamento de Tecnología Electrónica

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN EN VHDL DE UN DECODIFICADOR VITERBI Y SU INTEGRACIÓN EN UN PROTOTIPO DE UN SISTEMA WIMAX

Autor: Miguel Viñé Viñuelas

Tutor: Dr. Enrique San Millán Heredia

Leganés, junio de 2012

Título: Implementación en VHDL de un Decodificador Viterbi y su Integración en un Prototipo de un Sistema WiMAX.
Autor: Miguel Viñé Viñuelas.
Director:

EL TRIBUNAL

Presidente: _____

Vocal: _____

Secretario: _____

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día __ de _____ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

Agradecimientos

A los que más me han aguantado durante este largo proyecto: mi hermano Jorge y mi tío Gabriel.

Resumen.

En los sistemas de comunicación se producen errores que hacen que la secuencia recibida por el receptor no sea igual a la transmitida. Por este motivo se emplean sistemas FEC, forward error correction. Estos sistemas consisten en un codificador-decodificador que permiten corregir los errores que añade el canal de transmisión. Los más habituales son los códigos convolucionales, los de bloque y los turbo códigos.

En este proyecto diseñamos, implementamos, simulamos y verificamos un decodificador Viterbi en hardware, con código VHDL. Se trata del algoritmo de máxima verosimilitud para decodificar un código convolucional. Por este motivo es muy utilizado como método de corrección de errores en los sistemas FEC.

En la Universidad estamos desarrollando un prototipo de un sistema WiMAX entre varios proyectos fin de carrera diferentes. Uno de los bloques de este protocolo WiMAX es un decodificador Viterbi. De manera que el trabajo en este proyecto consiste en diseñar un decodificador Viterbi, con las especificaciones indicadas por el protocolo WiMAX. Una vez implementado, simulado y verificado lo integraremos en el prototipo.

FEC corrige los errores en el receptor sin retransmitir la secuencia original. Se utiliza en sistemas sin reenvío, o en sistemas en tiempo real donde no se puede esperar a la retransmisión para mostrar los datos. El proceso consiste en codificar la secuencia original, añadiéndole bits redundantes antes de transmitirla. La secuencia codificada redundante se transmite y llega al receptor. En el receptor hay un decodificador que obtiene la secuencia más próxima a la original antes de la codificación. Con este sistema se disminuye la BER en la salida del receptor, sin necesidad de aumentar Eb/No.

El decodificador Viterbi se inventó en el año 1967, y se emplea en muchos sistemas de comunicación, donde los datos transmitidos son propensos a errores antes de la recepción. Es compatible con multitud de estándares habituales, por ejemplo: GSM, WLAN, IS-54, IS-95, CDMA, ADSL, SHDSL, HDSL2, DVB, 3G, 3GPP, 3GPP2, 3GPP LTE, IEEE 802.16, HiperLAN, Intelsat IESS-308/309, IEEE 802.11a, DBS, VSAT... Por tanto, obviamente ya existen dispositivos que lo implementan en hardware. Se trata de IP (intellectual property) cores, que están disponibles en el mercado, pero no son libres, se requiere una licencia para utilizarlos.

El decodificador Viterbi es un algoritmo estándar, con diversos parámetros determinados por unas especificaciones. Estos parámetros permiten ajustar la decodificación a los requisitos del sistema de comunicaciones. Lógicamente, con unas especificaciones complejas se consigue una mayor reducción de la BER que con unas especificaciones simples, pero a cambio el decodificador será más complejo.

Una vez fijadas las especificaciones del decodificador, todos los dispositivos que las cumplan implementan el mismo algoritmo. Por tanto, no existirán diferencias de funcionalidad entre ellos. Este es un aspecto importante, que nos indica que nuestro decodificador debe tener la misma funcionalidad y los mismos puertos de entrada-salida, que cualquier IP Viterbi decoder que cumpla las mismas especificaciones.

Como en todo diseño hardware, optimizaremos el código para obtener el mejor compromiso entre mínima área ocupada y máxima frecuencia de reloj.

Nuestro objetivo principal ha sido diseñar un decodificador Viterbi que cumpla exactamente el algoritmo. Por tanto es funcionalmente igual a cualquier modelo que ya exista en el mercado. Sin embargo, hemos dado un valor añadido a nuestro decodificador, que supone una ventaja frente a la mayoría de los decodificadores comerciales. Consiste en que lo hemos realizado completamente multiplataforma, independiente de la tecnología. Nuestro código VHDL se puede implementar en cualquier FPGA de cualquier fabricante. Y también en el resto de dispositivos lógicos programables, por ejemplo los más comunes: ASICs, CPLDs, PLDs... Esto es una ventaja frente a la mayoría de módulos comerciales, que suelen ser específicos para una arquitectura concreta, son dependientes de la tecnología. Por ejemplo, los fabricantes de FPGAs: Xilinx, Actel, Lattice y Altera disponen de un IP core Viterbi decoder. Pero sólo puede utilizarse en determinadas FPGAs de su propio catálogo de productos.

Una parte fundamental es la verificación, para ello hemos desarrollado simuladores que nos permiten comparar nuestro decodificador con otro que utilizamos como referencia: el Xilinx IP core Viterbi decoder. Hemos realizado una simulación exhaustiva, gracias a la cual podemos asegurar que nuestro decodificador implementa exactamente el algoritmo Viterbi, sin ningún error de funcionalidad. A continuación, una vez finalizada la verificación, integraremos el decodificador en el prototipo WiMAX, finalizando así el proyecto.

Palabras clave.

WiMAX: Worldwide Interoperability for Microwave Access. (Interoperabilidad mundial para acceso por microondas).
Decodificador Viterbi, codificador convolucional, BER (bit error rate), Eb/No, simulador, FEC, VHDL, multiplataforma, verificación, FPGA (field programmable gate array), IP core (módulo de propiedad intelectual).

Abstract.

In communication systems are errors that cause the sequence at the receiver is not the same to the sequence transmitted. For this reason, is usual to use FEC, forward error correction, systems. These systems consist of an encoder-decoder that make possible to correct the errors added by the transmission channel. The most common are convolutional codes, block codes and turbo codes.

In this project we design, implement, simulate and verify a Viterbi decoder in hardware with VHDL code. It is the maximum likelihood algorithm for decoding a convolutional code. For this motive it is widely used as method to correct errors in FEC systems.

At the University we are developing a prototype of a WiMAX system between several different final degree projects. One of the blocks of this WiMAX protocol is a Viterbi decoder. So that, the work on this project is to design a Viterbi decoder, with the specifications indicated by the WiMAX protocol. Once implemented, simulated and verified, we will integrate it in the prototype.

FEC is a mechanism to correct mistakes, that allows to correct them in the receiver without retransmitting the original sequence. It is useful in systems without a reverse channel to request retransmission of data, or in real time systems where it is not possible to wait for the retransmission to show the information. The process consists in coding the original sequence, adding redundant bits before transmitting it. The coded redundant sequence is transmitted and arrives to the receiver. In the receiver there is a decoder that obtains the sequence closest to the original one before the codification. With this system the BER is decreased in the output of the receiver, without need to increase Eb/No.

The Viterbi decoder was invented in year 1967, and it is used in many communication systems, where the transmitted data are prone to errors before the reception. It is compatible with many habitual standards, for example: GSM, WLAN, IS-54, IS-95, CDMA, ADSL, SHDSL, HDSL2, DVB, 3G, 3GPP, 3GPP2, 3GPP LTE, IEEE 802.16, HiperLAN, Intelsat IESS-308/309, IEEE 802.11a, DBS, VSAT... Therefore, obviously already exist devices that implement it in hardware. These devices are called IP (intellectual property) cores, and are available on the market, but they are not free, a license is required to use them.

The Viterbi decoder is a standard algorithm, with various parameters determined by specifications. These parameters allow to adjust decoding to the requirements of the communications system. Obviously, a complex specification achieves greater reduction of BER that with a simple specification, but in return the decoder will be more complex.

Once laid down the specifications of the decoder, all devices that fulfill them implement the same algorithm. Therefore, there will not exit differences of functionality between them. This is an important aspect, which tell us that our decoder must have the same functionality and the same input-output ports, as any IP Viterbi decoder that fulfills the same specifications.

As in all hardware design, we will optimize the code for getting the best compromise between minimum occupied area and maximum clock frequency.

Our main aim has been to design a Viterbi decoder that fulfills exactly the algorithm. Therefore it is functionally equal to any model which already exists on the market. However, we have given an added value to our decoder, which is an advantage over the majority of commercial decoders. It consists that we have realized it fully multi-platform, technology independent. Our VHDL code can be implemented onto any FPGA from any manufacturer. And also onto the rest of programmable logic devices, for example the most common: ASICs, CPLDs, PLDs... This is an advantage over the majority of commercial devices, which are usually specific for a concrete architecture, they are technology dependent. For example, the FPGA manufacturers: Xilinx, Actel, Lattice and Altera have a IP core Viterbi decoder. But it can only be used onto certain FPGAs of its own catalog of products.

A fundamental part is the verification, for this, we have developed simulators that allow us to compare our decoder with another that we use as a reference: the Xilinx IP core Viterbi decoder. We have carried out an exhaustive simulation, whereby we can assure that our decoder implements exactly the Viterbi algorithm, without any functionality error. Then, once the verification has been completed, we will integrate the decoder in the prototype WiMAX, finishing this way the project.

Keywords.

WiMAX: Worldwide Interoperability for Microwave Access.
Viterbi decoder, convolutional encoder, BER (bit error rate), Eb/No, simulator, FEC, VHDL, multi-platform, verify, FPGA (field programmable gate array), IP (intellectual property) core.

Índice general.

1. INTRODUCCIÓN Y OBJETIVOS 1
1.1 Introducción. 2
1.1.1 Introducción al prototipo WiMAX. 2
1.1.2 Introducción a la decodificación Viterbi. 3
1.2 Objetivos. 4
1.3 Fases de desarrollo. 6
1.3.1 División del trabajo: decodificador de Opencores y UC3M. 9
1.4 Especificaciones. 11
1.4.1 Medios hardware y software empleados. 13
1.5 Estructura de la memoria. 15
1.6 VHDL. 17
1.7 Diseño hardware multiplataforma. 17
1.8 Flujo de diseño en una FPGA. 20
1.8.1 Pasos para implementar el diseño en un hardware diferente. 23
1.9 Referencias 24
1.9.1 VHDL. 24
1.9.2 System Generator. 25
1.9.3 WiMAX. 26
1.9.4 Tarjeta Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 xc4vsx55. 27
1.9.5 Xilinx LogiCore IP Viterbi decoder v7.0 28
1.9.6 Referencias genéricas. 29

2. CARACTERÍSTICAS CODIFICADOR CONVOLUCIONAL, DECODIFICADOR VITERBI. 32
2.1 Objetivos. 33
2.2 Reseña histórica, ¿quién es Viterbi? 33
2.3 Alternativas para controlar los errores en un sistema de telecomunicación. 35
2.3.1 Diagrama de bloques de un sistema de comunicaciones FEC 37
2.4 Codificación convolucional. 41
2.4.1 Introducción. 41
2.4.2 Estructura básica, parámetros que definen al codificador. 43
2.4.3 Codificadores no recursivos (FIR) o recursivos (IIR). 46
2.4.4 Codificadores sistemáticos o no sistemáticos. 46
2.4.5 Notación de la función codificadora. 46
2.4.6 Ejemplos de codificadores con diferentes características. 47
2.4.7 Maquina de estados. 54
2.5 Decodificación Viterbi. Fundamentos teóricos. 58
2.5.1 Objetivos. 58
2.5.2 Codificación-decodificación, sistema FEC. 59
2.5.3 Características decodificador Viterbi. 60
2.6 Arquitectura y funcionamiento decodificador Viterbi. 64
2.6.1 Caso general empleando intercambio de registros. 66
2.6.2 Inicio de la trama de entrada empleando intercambio de registros. 76
2.6.3 Final de la trama de entrada empleando truncamiento de trellis. 77
2.6.4 Método traceback. Comparativa con intercambio de registros. 78
2.7 Ejemplo práctico de codificación-decodificación con 4 estados. 80
2.7.1 Inicio de la trama de entrada. 81

2.7.2 Caso general. 84
2.7.3 Final de la trama de entrada. Resultados. 87
2.8 Referencias. 88
2.8.1 Genéricas sobre codificación convolucional-decodificación Viterbi. ... 88
2.8.2 Ejemplo 4 estados. 91
2.8.3 Arquitectura completa decodificador Viterbi. 91
2.8.4 ACS 92
2.8.5 Intercambio de registros y método traceback. 92
2.8.6 Otras referencias. 93

3. IMPLEMENTACIÓN CODIFICADOR CONVOLUCIONAL: EncoderK7.vhd 95
3.1 Características. 96
3.2 Interfaz entrada/salida. Uso del codificador. 97
3.2.1 Definición de puertos. 100
3.3 Arquitectura del codificador. 101
3.4 Verificación de que funciona correctamente. 103
3.4.1 Pasos a seguir. 103
3.4.2 Simulación usando como referencia ConvolutionalEncoder.m 104
3.4.3 Verificación comparando con Xilinx IP core convolutional encoder. 105
3.5 Resultados síntesis. Comparativa con el IP core de Xilinx. 107
3.6 Referencias. 108

4. IMPLEMENTACIÓN DECODIFICADOR OPENCORES: decoderverilog.v 110
4.1 Objetivos decodificador de Opencores. 111
4.2 Pasos seguidos en el proyecto. 113
4.3 Características. 115
4.4 Interfaz Entrada/Salida. Uso del decodificador. 116
4.4.1 Definición puertos. 119
4.5. Obtención decoderverilog.v. 120
4.5.1 Obtención del código a partir del generador automático Opencores... 120
4.5.2 Añadidos al código obtenido automáticamente. 122
4.6 Errores. 123
4.6.1 No se trata exactamente del algoritmo de Viterbi. 123
4.6.2 Demostración del error utilizando un ejemplo con 4 estados. 124
4.6.3 Hay latches. 127
4.7 Comparación con un decodificador Viterbi exacto. 128
4.8 Referencias. 129
4.8.1 Verilog. 130

5. IMPLEMENTACIÓN DECODIFICADOR UC3M: ViterbiDecoder.vhd 131
5.1 Características. 132
5.2 Interfaz Entrada/Salida. Uso del decodificador. 133
5.2.1 Definición puertos. 135
5.3 Codificación del estado i y de sus anteriores j y h. 136
5.4 Arquitectura y funcionamiento del sistema. 140
5.4.1 Ejemplo con 4 estados. 142
5.4.2 Descripción de ACS mediante el ejemplo. 144
5.4.3 Descripción de RegisterExchange mediante el ejemplo. 145
5.5 Descripción bloques. 147
5.5.1 Puertos comunes. 147

5.5.2 ACS.vhd. 149
5.5.3 UpdateStateInit.vhd. 154
5.5.4 DistanceJIandHI.vhd. 159
5.5.5 DistanceLastStateBitIn.vhd. 161
5.5.6 Normalize.vhd. 164
5.5.7 FindMinIndex.vhd. 167
5.5.8 Comparator4IN.vhd. 170
5.5.9 RegisterExchange.vhd. Estructura. 171
5.5.10 RegisterExchange. Funcionamiento con ejemplo de 4 estados. 176
5.5.11 CalculatePaths.vhd. 182
5.5.12 MemoriaRAM.vhd. 191
5.5.13 ExitDatoOut.vhd. 193
5.6 Referencias. 196

6. IMPLEMENTACIÓN SIMULADORES. 197
6.1 Objetivos. 198
6.2 Simuladores realizados. 199
6.2.1 Simuladores codificados con VHDL. 199
6.2.2. Simuladores codificados con System Generator. 200
6.3 Arquitectura básica. 202
6.4 Modelado del ruido. 205
6.4.1 Ecuaciones matemáticas. 205
6.4.2 Implementación del ruido uniforme en el simulador. 208
6.4.3 Implementación errores de ráfaga. 209
6.4.4 Gráficas BER en función de Eb/No. 211
6.5 Arquitectura simuladores en VHDL. 216
6.5.1 MantienePulso.vhd. 217
6.5.2 Retardador.vhd. 219
6.6 Manual de usuario Simuladores en VHDL. 221
6.6.1 Resultados. 221
6.6.2 Configuración simulador. 223
6.7 Arquitectura simuladores en System Generator. 226
6.7.1 Archivos implementados. 227
6.7.2 Añadidos para adaptarse a la técnica cola de ceros. 227
6.7.3 Descripción funcionamiento. 228
6.8 Manual de usuario Simuladores en System Generator. 229
6.8.1 Resultados. 229
6.8.2 Configuración simulador. 232
6.9 Referencias. 235

7. COMPARATIVA DECODIFICADORES. RESULTADOS SÍNTESIS Y SIMULACIÓN. 237
7.1 Objetivos. 238
7.2 Síntesis UC3M, ViterbiDecoder.vhd. 239
7.2.1 Resumen resultados tras síntesis y map. 239
7.2.2 Elección de arquitectura combinada, 8 etapas serie*8 módulos paralelo. 240
7.2.3 Optimización frecuencia máxima. 241
7.2.4 Optimización área. 242
7.2.5 Modificación decoding depth. 244

7.2.6 Modificación parámetros decodificador. ... 245
7.3 Síntesis UC3M, SimulacionCompleta.vhd ... 245
7.4 Síntesis Opencores, decoderverilog.v. ... 247
7.5 Comparativa de latencias en los 3 decodificadores. ... 248
7.6 Síntesis Xilinx IP core Viterbi decoder 7.0. ... 249
7.7 Gráficas BER frente a Eb/No. ... 251
7.7.1 Verificación de que el UC3M es funcionalmente correcto. ... 251
7.7.2 Gráficas decodificador UC3M frente a Xilinx. ... 255
7.7.3 Efectos al variar el decoding depth. ... 258
7.7.4 Dependencia con la longitud de la trama. ... 261
7.7.5 Gráficas decodificador UC3M frente a Opencores. ... 263
7.8 Comportamiento ante errores de ráfaga. ... 264
7.9 Referencias ... 267

8. INTEGRACIÓN ViterbiDecoder.vhd EN EL SISTEMA WIMAX. ... 272
8.1 Objetivos. ... 273
8.2 Características sistema WiMAX. ... 274
8.3 Integración de ViterbiDecoder.vhd en el sistema WiMAX. ... 275
8.4 Referencias. ... 277

9. PRESUPUESTO. ... 278
9.1 Objetivos. ... 279
9.2 Elaboración presupuesto. ... 280
9.2.1 Coste en horas de ingeniería. ... 280
9.2.2 Costes materiales. ... 282
9.2.3 Costes software. ... 283
9.3 Conclusiones. ... 284

10. CONCLUSIONES. ... 285
10.1 Diseño y verificación de ViterbiDecoder y EncoderK7. ... 286
10.2 Características ViterbiDecoder.vhd ... 289
10.3 Características decoderverilog.v. ... 291
10.4 Trabajos futuros. ... 292

A. LISTADO CÓDIGO FUENTE Y SIMULADORES. ... 296
A.1 Organización de los diseños en proyectos y carpetas. ... 297
A.2 Archivos principales. ... 298
A.2.1 Módulos codificador y decodificador. ... 298
A.2.2 Simuladores realizados con código VHDL. ... 298
A.2.3 Simuladores realizados con System Generator. ... 300
A.2.4 Modificaciones para variar la profundidad de memoria. ... 301
A.3 Código fuente VHDL y Verilog. ... 302
A.3.1 Decodificador y codificador UC3M. ... 302
A.3.2 Decodificador Opencores. ... 304
A.3.3 Simuladores y estructura UUTs. ... 305
A4. código fuente System Generator. ... 307
A.5 Ficheros de texto simuladores VHDL. ... 308
A6. Verificación de que EncoderK7.vhd es funcionalmente correcto. ... 309
A.7 Test bench para el cálculo de estados anteriores. ... 310

B. DESARROLLO MATEMÁTICO CODIFICADOR RECURSIVO. 312
B1 Codificador convolucional(2, 1, m=2), sistemático recursivo. 313
B2 Codificador convolucional(3, 2, m=3), sistemático recursivo. 314

C. FUNDAMENTOS MATEMÁTICOS DECODIFICADOR VITERBI 315

D. SÍNTESIS DE TODOS LOS MÓDULOS VHDL 319
D.1 Objetivos. 320
D2 Síntesis codificador convolucional EncoderK7.vhd. 321
D3 Síntesis ViterbiDecoder.vhd con profundidad de memoria 36. 322
D4 síntesis SimulacionCompleta.vhd. 332
D5 Síntesis SimulacionCompletaVerilog.vhd. 335
GLOSARIO. 337

Índice de figuras.

Figura 1.1: Pasos seguidos en el proyecto. 10
Figura 1.2: Creación de un IP core en Xilinx ISE8.1. 18
Figura 1.3: Ejemplos de IP cores disponibles en Xilinx ISE 8.1. 19
Figura 1.4: Flujo de diseño en una FPGA. 20

Figura 2.1 Diagrama de bloques sistema FEC. 37
Figura 2.2: Codificador convolucional(2, 1,K=3), no sistemático, no recursivo. 48
Figura 2.3: Codificador convolucional(2 , 1 ,K=7), no sistemático, no recursivo. 49
Figura 2.4: Codificador convolucional(3, 2, K=3), no sistemático, no recursivo. 50
Figura 2.5: Codificador convolucional(3, 2, K=2), sistemático, no recursivo. 51
Figura 2.6: Codificador convolucional(2, 1, m=2), sistemático, recursivo. 52
Figura 2.7: Codificador convolucional(3, 2, m=3), sistemático, recursivo. 53
Figura 2.8: Diagrama de estados codificador ejemplo 1 (2, 1,K=3). 57
Figura 2.9: Malla trellis codificador ejemplo 1 (2, 1,K=3). 57
Figura 2.10: Decodificador Viterbi en un sistema FEC. 59
Figura 2.11: Cuantificación empleando 1 y 2 bits. 62
Figura 2.12: Arquitectura genérica decodificador Viterbi. 64
Figura 2.13: Cálculo de Distance$_i$[t_{x+1}] 66
Figura 2.14: Arquitectura módulo ACS. 68
Figura 2.15: Situación de los caminos en t_6, al inicio del período de proceso. 71
Figura 2.16: Trabajo del decodificador en el período de proceso entre t_6 y t_7. 72
Figura 2.17: Situación de los caminos en t_7, final del período de proceso. 72
Figura 2.18: Ejemplo de decodificación entre n=0 y n=3. 81
Figura 2.19: Ejemplo de decodificación entre n=3 y n=4. 82
Figura 2.20: Ejemplo de decodificación entre n=4 y n=5. 83
Figura 2.21: Ejemplo de decodificación entre n=5 y n=6. 84
Figura 2.22: Ejemplo de decodificación entre n=6 y n=7. 85
Figura 2.23: Ejemplo de decodificación entre n=7 y n=8. 86

Figura 3.1: Interfaz del codificador: EncoderK7.vhd. 97
Figura 3.2: Arquitectura codificador convolucional UC3M: EncoderK7.vhd. 101
Figura 3.3: Estructura exacta de los registros. 101
Figura 3.4: Función de transferencia del codificador convolucional. 102
Figura 3.5: Sistema para verificar que EncoderK7.vhd es funcionalmente correcto. .. 104
Figura 3.6: Esquema completo de ComparaEncoderUC3M_Vs_EncoderXilinx.mdl. 106
Figura 3.7: Detalle del simulador ComparaEncoderUC3M_Vs_EncoderXilinx.mdl.. 106

Figura 4.1: Desarrollo del proyecto. 113
Figura 4.2: Interfaz del decodificador. decoderverilog.v. 116
Figura 4.3: Malla trellis completa, ejemplo 4 estados. 125
Figura 4.4: Extracción de los bits decodificados en el modelo de Opencores. 126
Figura 4.5: Extracción del bit decodificado en un algoritmo Viterbi exacto. 126
Figura 4.6: BER(Eb/No). Decodificadores Opencores y UC3M, decoding depth=32. 128

Figura 5.1: Interfaz del decodificador. ViterbiDecoder.vhd. 133
Figura 5.2: Codificador convolucional. K=7. 136
Figura 5.3 : Representación de los estados posteriores a i. 137
Figura 5.4 : Representación de los estados anteriores a i. 137
Figura 5.5: Arquitectura ViterbiDecoder.vhd. Nota 5.1. 140
Figura 5.6: Codificación del estado i=62 de la malla trellis. 141
Figura 5.7: Malla trellis del ejemplo con 4 estados 143
Figura 5.8A: Registros con todas sus señales. Igual para todos los módulos 148
Figura 5.8B: Registros con EnableIn en ACS, FindMinIndex y CalculatePaths. 148
Figura 5.9: ACS.vhd. 149
Figura 5.10: UpdateStateInit.vhd. 154
Figura 5.11: Función de UpdateStateInit en ACS. 155
Figura 5.12: Funcionamiento en ejemplo de 4 estados. 156
Figura 5.13: Actualización de InitialState en decodificador con 64 estados 158
Figura 5.14: DistanceJIandHI.vhd. 159
Figura 5.15: Trabajo de DistanceJIandHI y DistanceLastStateBitIn. 160
Figura 5.16: DistanceLastStateBitIn.vhd 161
Figura 5.17: Normalize.vhd. 164
Figura 5.18: Detalle de Normalize integrado en ACS. 166
Figura 5.19: FindMinIndex.vhd 167
Figura 5.20: Comparator4IN.vhd. 170
Figura 5.21: RegisterExchange.vhd. 171
Figura 5.22: Trellis caso general. 177
Figura 5.23: Trellis final de la trama. 180
Figura 5.24: CalculatePats.vhd 182
Figura 5.25: MemoriaRAM.vhd 191
Figura 5.26: ExitDatoOut.vhd. 193

Figura 6.1: Descripción sistema de comunicaciones. 202
Figura 6.2: Arquitectura básica simulador. 203
Figura 6.3: Efectos del ruido en el sistema. 205
Figura 6.4: Función densidad de probabilidad en el receptor. 206
Figura 6.5: Implementación del canal ruidoso. 209
Figura 6.6: Implementación errores de ráfaga. 210
Figura 6.7: Ejemplo de representación de la BER sin codificar y BERoutDecoder . .. 211

Figura 6.8: Representación ecuación característica BERinDecoder. 214
Figura 6.9. Arquitectura simuladores VHDL. 216
Figura 6.10: MantienePulso.vhd. 217
Figura 6.11: Retardador.vhd para decoding depth = 36 219
Figura 6.12: Consola del simulador VHDL. 221
Figura 6.13: Señales del simulador VHDL. 222
Figura 6.14. Arquitectura simuladores System Generator. 226
Figura 6.15: Consola del simulador System Generator. 229
Figura 6.16: Señales del simulador System Generator. 231
Figura 6.17: Activación y desactivación de aclr entre tramas consecutivas. 232

Figura 7.1: Comparativa latencias UC3M y Xilinx. 249
Figura 7.2: Comparativa decodificador UC3M y Xilinx. Decoding depth 36. 256
Figura 7.3: Comparativa decodificador UC3M y Xilinx. Decoding depth 48. 256
Figura 7.4: Comparativa decoding depth 60 frente a 24. 259
Figura 7.5: Variación en la BER al modificar decoding depth. Trama continua. 259
Figura 7.6: Variación en la BER al modificar decoding depth. Trama 2000 símbolos. 260
Figura 7.7: Variación en la BER al modificar decoding depth. Trama 200 símbolos. 260
Figura 7.8: Pérdidas de Eb/No al disminuir el tamaño de la trama de entrada. 261
Figura 7.9: Técnicas truncamiento de trellis y cola de ceros. 262
Figura 7.10: Rendimiento Opencores. 263
Figura 7.11: BER frente a Eb/No con errores de ráfaga. 265
Figura 7.12: Formato de la cadena de entrada con errores de ráfaga. 265

Figura 8.1: Xilinx IPcore Viterbi decoder v5.0 en el sistema original. 276
Figura 8.2: Sustitución del IP core de Xilinx por ViterbiDecoder.vhd. 276

Figura 10.1: Arquitectura UUT a emular (simulacioncompleta.bit). 294
Figura 10.2: Elementos necesarios para emular ViterbiDecoder.vhd. 295

Índice de tablas.

Tabla 2.1: Posibles representaciones de la relación entre la entrada y la salida..... 46
Tabla 2.2: Parámetros codificador figura 2.2. 48
Tabla 2.3: Parámetros codificador figura 2.3. 49
Tabla 2.4: Parámetros codificador figura 2.4. 50
Tabla 2.5: Parámetros codificador figura 2.5. 51
Tabla 2.6: Parámetros codificador figura 2.6. 52
Tabla 2.7: Parámetros codificador figura 2.7. 53
Tabla 2.8: Tabla estado/siguiente estado codificador ejemplo 1 (2, 1, K=3). 58
Tabla 2.9: Caminos supervivientes almacenados en la memoria en t_6 y t_7. 72
Tabla 2.10: Contenido de la memoria con método traceback, en t_6. 79
Tabla 2.11: Contenido de la memoria con método traceback, en t_7. 79
Tabla 2.12: Codificación secuencia ejemplo. Codificador (2, 1,K=3). 80
Tabla 2.13: Codificación secuencia ejemplo. Codificador (2, 1,K=3). 80
Tabla 2.14: Decodificación en el intervalo [n=0...n=3]. 81
Tabla 2.15: Decodificación en el intervalo [n=3...n=5]. 83
Tabla 2.16: Decodificación en el intervalo [n=5...n=8]. 86
Tabla 2.17: Decodificación del final de la trama..... 87

Tabla 3.1: Puertos EncoderK7.vhd..... 100
Tabla 3.2: Síntesis codificador convolucional UC3M y Xilinx 107
Tabla 3.3: Map codificador convolucional UC3M y Xilinx. 107

Tabla 4.1: Bits de relleno al final de la trama de entrada. 118
Tabla 4.2: Puertos decoderverilog.v 119
Tabla 4.3: Ejemplo 4 estados, bits en la entrada del decodificador. 124

Tabla 5.1: Puertos ViterbiDecoder.vhd 135
Tabla 5.2: MatrixLastStates..... 139
Tabla 5.3: Recopilación de la información del trellis..... 143
Tabla 5.4: Datos iniciales en el módulo CalculatePaths..... 145
Tabla 5.5: Datos finales en el módulo CalculatePaths. 146
Tabla 5.6: Proceso completo en ACS, etapa a etapa 153
Tabla 5.7: Proceso actualización InitialState. Ejemplo malla 4 estados. 157
Tabla 5.8: Proceso actualización InitialState. Malla 64 estados..... 158
Tabla 5.9: Funcionamiento FindMinIndex..... 167
Tabla 5.10: Información contenida en el trellis..... 177
Tabla 5.11: Entradas en RegisterExchange. 178
Tabla 5.12: Contenido de la memoria al llegar t_{x+1}. 179
Tabla 5.13: Información contenida en el trellis. 180
Tabla 5.14: Organización de la memoria RAM. 189
Tabla 5.15: Síntesis CalculatePaths.vhd. Comparativa RAM lectura síncrona/asíncrona. 192

Tabla 6.1: Puntos para la representación de BERinDecoder frente a Eb/No 214
Tabla 6.2: Latencia entre EnableInEncoder y EnableOutDecoder..... 220
Tabla 6.3: Valor mínimo en ACLR_Desactivado 233

Tabla 7.1: Síntesis ViterbiDecoder.vhd. (Decoding depth = 36) 239
Tabla 7.2: Map ViterbiDecoder.vhd. (Decoding depth = 36) 240
Tabla 7.3: Comparativa Síntesis CalculatePaths.vhd 242
Tabla 7.4: Síntesis ViterbiDecoder.vhd. Decoding depth 8, 24, 32, 36, 48, 60 244
Tabla 7.5: Síntesis SimulacionCompleta.vhd. (Decoding depth = 36) 246
Tabla 7.6: Map SimulacionCompleta.vhd. (Decoding depth = 36) 246
Tabla 7.7: Síntesis decoderverilog.v 247
Tabla 7.8: Latencias de todos los diseños. 248
Tabla 7.9: Síntesis Viterbi decoder v7.0 de Xilinx 250
Tabla 7.10: Puntos para representar una curva con errores de ráfaga. 267

Tabla 9.1: Fases del proyecto y horas dedicadas a las mismas. 281
Tabla 9.2: Gastos en material. 282
Tabla 9.3: Costes software 283

Tabla A1: Ficheros de texto empleados en los simuladores VHDL. 308
Tabla D1: Síntesis EncoderK7.vhd 321
Tabla D2: Map EncoderK7.vhd 321
Tabla D3: Síntesis ViterbiDecoder.vhd, profundidad de memoria=36. 323
Tabla D4: Map ViterbiDecoder.vhd, profundidad de memoria=36. 323
Tabla D5: Síntesis ACS.vhd, profundidad de memoria=36. 324
Tabla D6: Map ACS.vhd, profundidad de memoria=36. 324
Tabla D7: Síntesis RegisterExchange.vhd, profundidad de memoria=36. 324
Tabla D8: Map RegisterExchange.vhd, profundidad de memoria=36. 325
Tabla D9: Síntesis calculatePaths.vhd. 325
Tabla D10: Map calculatePaths.vhd. 326
Tabla D11: Síntesis ExitDatoOut.vhd. 326
Tabla D12: Map ExitDatoOut.vhd. 327
Tabla D13: Síntesis memoriaRAM.vhd 327
Tabla D14: Map memoriaRAM.vhd 327
Tabla D15: Síntesis FindMinIndex.vhd 328
Tabla D16: Map FindMinIndex.vhd 328
Tabla D17: Síntesis Comparator4IN.vhd 328
Tabla D18: Map Comparator4IN.vhd 329
Tabla D19: Síntesis UpdateStateInit.vhd 329
Tabla D20: Map UpdateStateInit.vhd 329
Tabla D21: Síntesis DistanceJIandHI.vhd 330
Tabla D22: Map DistanceJIandHI.vhd 330
Tabla D23: Síntesis DistanceLastStateBitIN.vhd 330
Tabla D24: Map DistanceLastStateBitIN.vhd 331
Tabla D25: Síntesis normalice.vhd 331
Tabla D26: Map normalice.vhd 331
Tabla D27: Síntesis SimulacionCompleta.vhd, profundidad de memoria 36. 332
Tabla D28: Map SimulacionCompleta.vhd, profundidad de memoria 36. 333
Tabla D29: Síntesis MantienePulso.vhd, cualquier profundidad de memoria. 333
Tabla D30: Map MantienePulso.vhd, cualquier profundidad de memoria. 334
Tabla D31: Síntesis Retardador.vhd, cualquier profundidad de memoria. 334
Tabla D32: Map Retardador.vhd, cualquier profundidad de memoria 334
Tabla D33: Síntesis SimulacionCompletaVerilog.vhd, profundidad de memoria 32. 335
Tabla D34: Map SimulacionCompletaVerilog.vhd, profundidad de memoria 32. 336

Índice de cronogramas.

Cronograma 3.1: Codificador UC3M, EncoderK7.vhd, efecto $\overline{aclr}$ 97
Cronograma 3.2: Codificador UC3M, codificación continua y en pulsos. 98

Cronograma 4.1: decoderverilog.v, inicio de la trama. Decoding depth=32. 116
Cronograma 4.2: decoderverilog.v, final de la trama. Decoding depth=32. 116

Cronograma 5.1: ViterbiDecoder.vhd, inicio de la trama. Decoding depth=36..... 133
Cronograma 5.2: ViterbiDecoder.vhd, final de la trama. Decoding depth=36. 133
Cronograma 5.3: ViterbiDecoder.vhd, intervalo entre tramas. Decoding depth=36.... 134
Cronograma 5.4: ACS.vhd Funcionamiento habitual. 150
Cronograma 5.5: ACS.vhd detalle inicio y final de la trama de entrada. 151
Cronograma 5.6: Normalize.vhd. NormalizeTime=50..... 165
Cronograma 5.7: RegisterExchange.vhd. Decoding depth=36. 175
Cronograma 5.8: CalculatePaths.vhd. Modo General. 187
Cronograma 5.9: CalculatePaths.vhd. Fin de la trama de entrada..... 188
Cronograma 5.10: ExitDatoOut.vhd..... 195

Cronograma 6.1: MantienePulso.vhd 218
Cronograma 6.2: Retardo en el sistema. 220

CAPÍTULO 1

1. INTRODUCCIÓN Y OBJETIVOS.

1.1 Introducción.

1.1.1 Introducción al prototipo WiMAX.

La necesidad de este proyecto surge porque en la Universidad Carlos III de Madrid estamos desarrollando un prototipo de un sistema WiMAX, cuyas especificaciones se detallan en el *apartado 1.4*. En este desarrollo hay dos departamentos trabajando: Tecnología Electrónica y Teoría de la Señal y Comunicaciones. Se está realizando entre varios proyectos fin de carrera diferentes, uno de ellos es el que describimos en esta memoria.

Uno de los bloques del sistema WiMAX es un decodificador Viterbi, con las especificaciones indicadas en el *apartado 1.4*. De manera que el objetivo principal de este proyecto, es desarrollar un decodificador Viterbi en hardware con código VHDL, estándar IEEE 1076-1993, que cumpla esas especificaciones. Realizaremos todo el proceso de: diseño, implementación, simulación y verificación. Y una vez que nos hayamos asegurado de que nuestro diseño es correcto y cumple las especificaciones requeridas, lo integraremos en el prototipo WiMAX.

Disponemos de una tarjeta Lyrtech VHS-ADC con una FPGA Virtex 4 xc4vsx55 10ff1148. El prototipo WiMAX se implementa en esa FPGA y su código está desarrollado con Xilinx System Generator for DSP. Nuestro decodificador lo implementaremos con código VHDL. De manera que debemos obtener una caja negra, con los mismos puertos de entrada-salida y el mismo funcionamiento, que el bloque decodificador Viterbi especificado por el WiMAX. Una vez finalizado el decodificador, y tras verificar que funciona correctamente, integraremos la caja negra en el prototipo.

WiMAX son las siglas de: Worldwide Interoperability for Microwave Access. (Interoperabilidad mundial para acceso por microondas). El protocolo utiliza la norma definida por el estándar IEEE Std 802.16^{TM} –2004. "IEEE Standard for Local and metropolitan area networks". Disponemos de esta norma completa en la referencia [22].

Un sistema WiMAX permite una comunicación inalámbrica de banda ancha entre un emisor y un receptor. Se pueden alcanzar distancias de hasta 50 Km si hay visibilidad directa entre la antena transmisora y receptora, (sistemas LOS Line of Sight). Y varios kilómetros en el caso de que no haya visibilidad directa entre las dos antenas, (Sistemas NLOS (Non Line of Sight). Esto lo diferencia de Wi-Fi, IEEE 802.11, donde se permiten distancias de varios metros en NLOS y de cientos de metros en LOS.

WiMAX es una alternativa a los sistemas cableados como xDSL y fibra óptica, ofreciendo unas prestaciones similares, pero a un coste inferior. Su aplicación más interesante consiste en dar servicios de banda ancha en zonas de baja densidad de población o de difícil acceso. Donde tender una red de cable o de fibra supone un coste por usuario muy elevado.

Para no extendernos demasiado en esta introducción no continuaremos describiendo el sistema WiMAX. Pero hemos seleccionado una amplia bibliografía en el *apartado 1.9.3* donde se puede ampliar información. Además en el *apartado 8.2* ampliamos la descripción sobre WiMAX.

1.1.2 Introducción a la decodificación Viterbi.

El decodificador Viterbi es el algoritmo de máxima verosimilitud para decodificar un código convolucional. Por este motivo es ampliamente utilizado como método de corrección de errores en los sistemas FEC, forward error correction. FEC es un sistema que permite tanto la corrección como la detección de errores en el receptor, sin retransmitir la información original.

Otra alternativa ampliamente utilizada es ARQ, Automatic Repeat Request, (petición automática de retransmisión). En este sistema se pueden detectar errores en el receptor, pero no corregirlos. De manera que cuando el receptor detecte un error en un mensaje, enviará una solicitud de retransmisión del mensaje al transmisor.

Para no extendernos demasiado en esta introducción, no entraremos en más detalles sobre estos dos métodos. En el *apartado 2.3* los desarrollamos ampliamente, y también se puede ampliar información en las referencias [51] y [52] en *1.9.6*.

De manera que diseñamos el decodificador ajustado a las especificaciones del *apartado 1.4*, y lo integraremos en el prototipo WiMAX. Pero un aspecto importante es que estas especificaciones no son exclusivas del protocolo WiMAX. Hay muchos estándares de comunicaciones en los que también se utiliza un decodificador Viterbi ajustado a las mismas especificaciones. Por tanto, es importante indicar que hemos diseñado un decodificador genérico, que no sólo puede utilizarse en el WiMAX. Sino que también puede emplearse en cualquier aplicación, en la que se necesite un decodificador Viterbi con las mismas especificaciones.

En el resumen, al comienzo de esta memoria, indicamos que se trata de un algoritmo muy utilizado, y nombramos algunos de los estándares más habituales con los que es compatible. Además hemos seleccionado una extensa bibliografía, donde se puede ampliar la información sobre estos estándares de comunicaciones. Referencias [53] a [58] y página 1 de [43].

Una vez implementado el decodificador, un trabajo fundamental es asegurarse de que no hemos cometido ningún error, y el bloque realiza exactamente la decodificación Viterbi. Para ello nos valemos de que es un algoritmo estándar. De manera que nuestro bloque debe tener el mismo funcionamiento, y los mismos puertos de entrada-salida, que cualquier IP Viterbi decoder con las mismas especificaciones.

Desarrollaremos un simulador con el que compararemos nuestro bloque con otro que tomamos como referencia: el IP core Viterbi decoder v5.0 de Xilinx. Realizaremos una simulación exhaustiva, aplicando a los dos módulos exactamente la misma cadena de entrada, y comparando en tiempo real sus dos salidas. Esto nos permite verificar que nuestro decodificador no tiene errores, implementa el algoritmo exacto, y se comporta igual que el IP core de Xilinx.

Como indicamos en el resumen, una ventaja de nuestro decodificador frente a la mayoría de los IP cores presentes en el mercado, es que es multiplataforma, (independiente de la tecnología). Nuestro código VHDL se podrá implementar en cualquier FPGA de cualquier fabricante. Y también en el resto de dispositivos lógicos programables, por ejemplo los más comunes: ASICs, CPLDs, PLDs... Lógicamente,

siempre que tanto la herramienta de síntesis como el hardware acepten VHDL IEEE 1076-1993. Esto es lo habitual y ocurrirá en casi todas las ocasiones, porque VHDL es un lenguaje estándar en la industria, aceptado por la gran mayoría de fabricantes y de dispositivos lógicos programables. Bibliografía sobre los distintos dispositivos hardware programables en [59], [60], [61] y [62].

Sin embargo, los dispositivos comerciales suelen ser específicos para una arquitectura hardware concreta, (son dependientes de la tecnología). De manera que se adquiere la licencia que permita instanciar el decodificador adecuado a las tarjetas que se vayan a usar en el diseño. Pero si las tarjetas iniciales se cambian por otras no compatibles con el decodificador, no podremos instanciarlo en ellas, aunque dispongamos de la licencia. Esto no ocurrirá con nuestro decodificador, porque es compatible con todas las tarjetas hardware programables, actuales, pasadas y las que se desarrollen en el futuro. Lógicamente, siempre que acepten VHDL norma IEEE 1076-1993, algo que como acabamos de indicar, ocurrirá en la gran mayoría de situaciones.

En el *apartado 1.9.5* y en [70] a [77], referenciamos los datasheets y manuales de usuario de algunos ejemplos de decodificadores Viterbi dependientes de la tecnología. Se trata de los IP cores de Xilinx, Actel, Lattice y Altera, que disponen de un IP Viterbi que sólo se puede sintetizar en determinadas FPGAs de su propia familia de productos.

1.2 Objetivos.

1. **El objetivo principal es: diseñar, implementar, simular y verificar un decodificador Viterbi en hardware con código VHDL. Con la misma funcionalidad y los mismos puertos de entrada-salida, que cualquier IP core Viterbi decoder ajustado a las especificaciones indicadas en el *apartado 1.4*. Para ello desarrollaremos el bloque *ViterbiDecoder.vhd*. Y una vez implementado, simulado y verificado, lo integraremos en el prototipo WiMAX realizado con System Generator.**

2. Daremos un valor añadido a nuestro decodificador al realizarlo totalmente multiplataforma.

3. Para realizar el proceso de simulación y verificación de todos los decodificadores realizados, implementaremos simuladores que modelan un sistema FEC. En este sistema se generarán unos bits aleatorios que se codificarán convolucionalmente. A la secuencia codificada se le añadirá ruido, y el resultado será la entrada del decodificador Viterbi. Para finalizar, los propios simuladores analizarán la salida del decodificador, determinando la BER de salida.

4. En el modelo del sistema FEC incluiremos todas las variables que influyen en el comportamiento del decodificador. De esta manera modelaremos con total exactitud su entorno real cuando se implemente en hardware. Estas variables son:
   - Ruido uniforme con amplitud variable.
   - Errores de ráfaga con duración e intervalo entre ráfagas variables.
   - Secuencia de datos aleatorios de entrada al codificador, distribuidos mediante tramas de longitud y espacio entre tramas variable, o mediante una secuencia continua (trama de longitud infinita).

5. Realizaremos un simulador con VHDL y otro con System Generator. Ambos modelan el mismo sistema FEC, pero diseñamos dos porque con cada uno de ellos buscamos objetivos diferentes. El simulador codificado con VHDL será multiplataforma. En cambio, el codificado con System Generator no será multiplataforma, pero añadirá algunas ventajas que facilitarán el proceso de simulación. Para optimizar el tiempo empleado en estas tareas, ambos simuladores tendrán la misma estructura y compartirán algunos bloques.

6. Un valor añadido al simulador codificado con System Generator, es que nos permitirá simular al mismo tiempo nuestro decodificador y el que utilizamos como referencia: el Xilinx IP core Viterbi decoder v5.0. De esta manera podremos verificar que ambos se comportan igual. Esto es fundamental para poder asegurar que nuestro decodificador no tiene ningún error de funcionalidad.

7. Como en todo diseño hardware, es necesario optimizar el código, obteniendo el mejor compromiso entre mínima área ocupada y máxima frecuencia de reloj. Optimizaremos todo el código que realicemos: decodificador Viterbi, codificador convolucional y simuladores, (tanto los realizados con VHDL como con System Generator).

8. Ampliaremos algunas especificaciones iniciales para mejorar el decodificador, otorgándole más flexibilidad. La especificación es que la profundidad de memoria debe ser de 36 símbolos de entrada. Pero nosotros haremos este parámetro variable, de manera que se pueda implementar un decodificador con cualquier profundidad de memoria, con sólo cambiar una constante. Aprovechando esta ventaja, realizaremos el proceso completo de desarrollo, implementación, simulación y verificación de 6 decodificadores, con profundidades de memoria: 8, 24, 32, 36, 48 y 60.

9. Documentaremos todos los resultados tras la síntesis y simulación, comparándolos con el Viterbi de Xilinx. En esta tarea es fundamental realizar las gráficas BER frente a Eb/No.

El objetivo principal es desarrollar el código para integrarlo en el WiMAX, pero en base a este objetivo principal se proponen los siguientes objetivos parciales.

a) Implementar un codificador convolucional en hardware con código VHDL, realizando todos los pasos necesarios: diseño, implementación, simulación y verificación. Además, aprovecharemos las ventajas de VHDL y también será independiente de la tecnología. Este codificador es necesario para modelar el sistema FEC, y por este motivo debemos asegurarnos de que lo hemos realizado correctamente. De manera que desarrollaremos otro simulador para verificar su funcionamiento, comparándolo con el Xilinx IP core convolutional encoder v3.0.

b) Realizaremos un estudio tecnológico completo sobre todos los aspectos relacionados con este proyecto: decodificación Viterbi, codificación convolucional, sistemas FEC, modelado del ruido, WiMAX, simulación, proceso de desarrollo en una FPGA, VHDL, System Generator... De manera que uno de los objetivos es mostrar en esta memoria los conocimientos adquiridos.

c) También debemos seleccionar una adecuada bibliografía, donde el lector podrá ampliar los conocimientos que considere oportunos. Hemos consultado una bibliografía muy extensa, por ello no referenciaremos toda la documentación. Seleccionaremos únicamente los documentos que a nuestro juicio explican mejor cada tema en concreto. De esta manera se facilita al lector la búsqueda de información.

d) El objetivo final es implementar el módulo *ViterbiDecoder.vhd*. Sin embargo también desarrollamos otro decodificador Viterbi: *decoderverilog.v*. Este segundo decodificador lo desarrollamos partiendo de un modelo de la Web de Opencores: http://opencores.org/. Y sobre él también realizaremos todo el proceso completo de implementación y simulación. Se trata de un módulo que sólo utilizaremos de manera temporal, en ningún momento lo realizamos con el objetivo de que sea el módulo final de este proyecto. El motivo por el que desarrollamos dos decodificadores Viterbi lo explicamos en el *apartado 1.3.1*.

1.3 Fases de desarrollo.

En este proyecto debemos realizar estas tareas:

1. Realizar un estudio teórico previo, en el que adquirimos los conocimientos necesarios para realizar el proyecto.

2. Diseñar un codificador convolucional y un decodificador Viterbi con VHDL.

3. Optimizarlos en área y velocidad.

4. Sintetizarlos e implementarlos en hardware.

5. Desarrollar un simulador con VHDL.

6. Verificar el codificador y el decodificador mediante simulación funcional y post place & route.

7. Integrar el codificador y el decodificador en System Generator.

8. Desarrollar un simulador con System Generator.

9. Mediante el simulador del punto anterior, verificamos el codificador y el decodificador tras integrarlos en System Generator. Además comparamos nuestro decodificador con el Xilinx IP core Viterbi decoder.

10. Integrar el decodificador en el prototipo WiMAX.

11. Elaborar esta memoria, describiendo: el trabajo realizado; los conocimientos adquiridos; y proporcionando las referencias bibliográficas más adecuadas para cada uno de los aspectos relacionados con este proyecto.

En cuanto hayamos adquirido la base teórica adecuada, podemos comenzar con el diseño del decodificador, el codificador, y su descripción hardware mediante VHDL. Al finalizar el hito 4 de la lista anterior, dispondremos de unos módulos que podrán implementarse sin ningún error en cualquier dispositivo hardware. Para ello, el código que desarrollemos debe cumplir lo siguientes requisitos:

- Optimización en área y frecuencia máxima de reloj.
- Cumplir todas las reglas de diseño síncrono.
- Debemos aseguramos de que no haya errores estructurales ni de diseño.
- Debemos verificar que el circuito puede sintetizarse en hardware sin ningún error y es independiente de la tecnología.

El siguiente paso es asegurar que el circuito que hemos descrito tiene la funcionalidad necesaria. Para ello desarrollamos un simulador con código VHDL que modela un sistema FEC. Modificaremos las variables del simulador, hasta cubrir todas las situaciones posibles que influyen en el comportamiento del codificador y el decodificador. Al finalizar el proceso de simulación, habremos verificado que no hay errores funcionales. De manera que podremos asegurar que los bloques que hemos implementado realizan exactamente la función de un codificador convolucional y un decodificador Viterbi. Este trabajo se realiza mediante los hitos 5 y 6.

Al finalizar el paso 6 se habrá concluido correctamente el objetivo principal de: diseño, implementación, simulación y verificación de un decodificador Viterbi.

Entonces, la siguiente tarea consiste en integrar este decodificador Viterbi en un diseño realizado con System Generator, paso 7. Para su posterior integración en el prototipo WiMAX

Desarrollamos otro simulador, codificado con System Generator, para simular y verificar el decodificador Viterbi integrado en System Generator. Esto se realiza en los pasos 8 y 9, donde se busca el mismo objetivo que en los hitos 5 y 6. Por tanto, hay una duplicidad de acciones que suponen un tiempo añadido en el desarrollo del proyecto. Sin embargo, como veremos a continuación, los beneficios que obtenemos con esta duplicidad justifican sobradamente el tiempo empleado en su realización.

a) En primer lugar desarrollamos el modelo codificado con VHDL, para conseguir el objetivo de un simulador multiplataforma.

b) A continuación realizamos un segundo simulador con System Generator, que aporta algunas ventajas respecto al de VHDL. Sin embargo, tiene el inconveniente de que no es multiplataforma, porque se describe con un lenguaje dependiente de la tecnología de Xilinx. Este es el motivo fundamental que nos obliga a desarrollar dos simuladores.

c) Para optimizar el tiempo de desarrollo, todos los simuladores comparten la misma estructura. Además algunos de los bloques del modelo realizado con System Generator no hay que desarrollarlos, porque se importan del modelo codificado con VHDL, sin realizar cambios ni añadidos. Por tanto, conseguimos disminuir el tiempo empleado en diseño y codificación del modelo con System Generator.

d) Ambos simuladores tienen la misma funcionalidad y realizan el mismo modelo del sistema FEC, con las mismas variables. Sin embargo, el realizado con System Generator aporta 5 ventajas:

1) Permite simular al mismo tiempo nuestro decodificador y el que utilizamos como referencia: el Xilinx IP core Viterbi decoder v5.0. Se aplican a ambos las mismas señales de entrada y al mismo tiempo. A continuación se comparan sus salidas en tiempo real. Como los dos decodificadores implementan el mismo algoritmo, sus resultados de salida en términos de BER frente a Eb/No deben ser iguales. De esta manera podremos verificar que nuestro decodificador tiene el mismo funcionamiento que el core de Xilinx.

2) También hemos realizado otro modelo con System Generator que simula al mismo tiempo nuestro codificador y el que utilizamos como referencia: el Xilinx IP core convolutional encoder v3.0. Sin embargo, en el simulador codificado con VHDL no se puede incluir ningún core de Xilinx, porque entonces dejaría de ser independiente de la tecnología.

3) Toda la información de interés se monitoriza y se actualiza instantáneamente en tiempo real durante el transcurso de la simulación. Sin embargo, en el codificado con VHDL se monitoriza la misma información, pero no se actualiza hasta que haya finalizado el tiempo fijado para la simulación. Por tanto, en el modelo realizado con System Generator disponemos en todo momento de toda la información de interés, lo que permite un mejor control del proceso de simulación. Además, en algunas ocasiones dispondremos de suficientes resultados antes de que haya finalizado la simulación, por lo que podremos pararla antes. Sin embargo, en el modelo codificado con VHDL, siempre es necesario esperar hasta que termine el tiempo fijado.

4) El prototipo WiMAX trabaja con frecuencia F_{CLK}, mientras que el decodificador Viterbi trabaja con $8*F_{CLK}$. De manera que al integrar el decodificador en el prototipo WiMAX, es necesario adaptar las frecuencias de reloj, incluyendo bloques Up Sample y Down Sample. Esta misma adaptación la realizamos en el simulador codificado con System Generator. De manera que implementamos un modelo que incluye los mismos bloques hardware que rodearán al decodificador cuando se integre en el WiMAX.

5) Aprovechamos las características que ofrece Simulink para mejorar la interfaz gráfica y facilitar el manejo del simulador.

Estas ventajas justifican el desarrollo de un segundo simulador codificado con System Generator, porque el proceso de simulación es fundamental y ha requerido más de 3000 horas de computación. Por tanto, consideramos que dedicar tiempo para mejorar las herramientas empleadas es una medida muy acertada.

Un aspecto a tener en cuenta es que los pasos 6 y 9 son iguales, porque se realiza la misma función sobre el mismo decodificador. De manera que estos pasos no hay que repetirlos, sino que se complementan uno con otro. Porque con los dos simuladores se obtiene el mismo resultado.

Al finalizar el hito 9, el decodificador está finalizado y verificado. Sólo queda integrarlo en el prototipo WiMAX. Este paso es inmediato, lo único que hay que hacer es instanciar el decodificador como una caja negra, adaptando su frecuencia de reloj a la del prototipo WiMAX, mediante bloques Up Sample y Down Sample.

1.3.1 División del trabajo: decodificador de Opencores y UC3M.

Nuestro objetivo principal es implementar un decodificador Viterbi partiendo de cero, debe ser un desarrollo totalmente propio. Lo denominamos *ViterbiDecoder.vhd* y es el decodificador UC3M, el módulo final del proyecto.

Sin embargo, por causas ajenas al proyecto tuvimos que hacer un cambio en la planificación inicial. Porque fue necesario disponer de un decodificador Viterbi en un plazo muy breve de tiempo. El plazo no era suficiente para finalizar el módulo que estábamos desarrollando: *ViterbiDecoder.vhd*.

Para acortar el tiempo de desarrollo, lo que hicimos fue partir de un código libre disponible en Internet. En una página Web especializada, que ofrece gratuitamente módulos hardware codificados en VHDL y Verilog: http://opencores.org/. Esta página nos proporciona un decodificador Viterbi que es del que hemos partido. Sobre este código realizamos los cambios necesarios para ajustarlo a las especificaciones requeridas y a System Generator, obteniendo el decodificador de Opencores: *decoderverilog.v*.

Implementamos un simulador, codificado tanto con VHDL como con System Generator, para simular decoderverilog.v. Una vez finalizado todo el proceso de simulación, decoderverilog.v está listo para integrarse en el WiMAX.

Un requisito para todos los simuladores realizados, es que nos sirvan para cualquier decodificador. De esta manera todos ellos son válidos para los 3 decodificadores que utilizamos: el modelo de Opencores (*decoderverilog.v*), el UC3M (*ViterbiDecoder.vhd*), y el que utilizamos como referencia: Viterbi decoder v5.0 de Xilinx. Así conseguimos que la tarea de simular decoderverilog.v no suponga coste de desarrollo en horas de ingeniería. El coste es únicamente el tiempo de computación necesario para ejecutar las simulaciones.

Un aspecto importante es que el desarrollo de decoderverilog.v no es suficiente para cumplir con los objetivos del proyecto. El motivo fundamental es que no es un desarrollo propio. Otro motivo es que descubrimos que no implementa exactamente el algoritmo Viterbi, y tiene errores de arquitectura hardware. Por todas estas razones, una vez finalizada la simulación de decoderverilog.v y su posterior integración en el WiMAX, continuamos con nuestro diseño: ViterbiDecoder.vhd.

Una puntualización importante es que no hay ninguna relación entre los dos módulos. ViterbiDecoder.vhd es un diseño completamente propio, no depende en nada de decoderverilog.v, ni de ningún otro decodificador.

En la siguiente figura, se aprecian claramente las fases que han sido necesarias para realizar el proyecto:

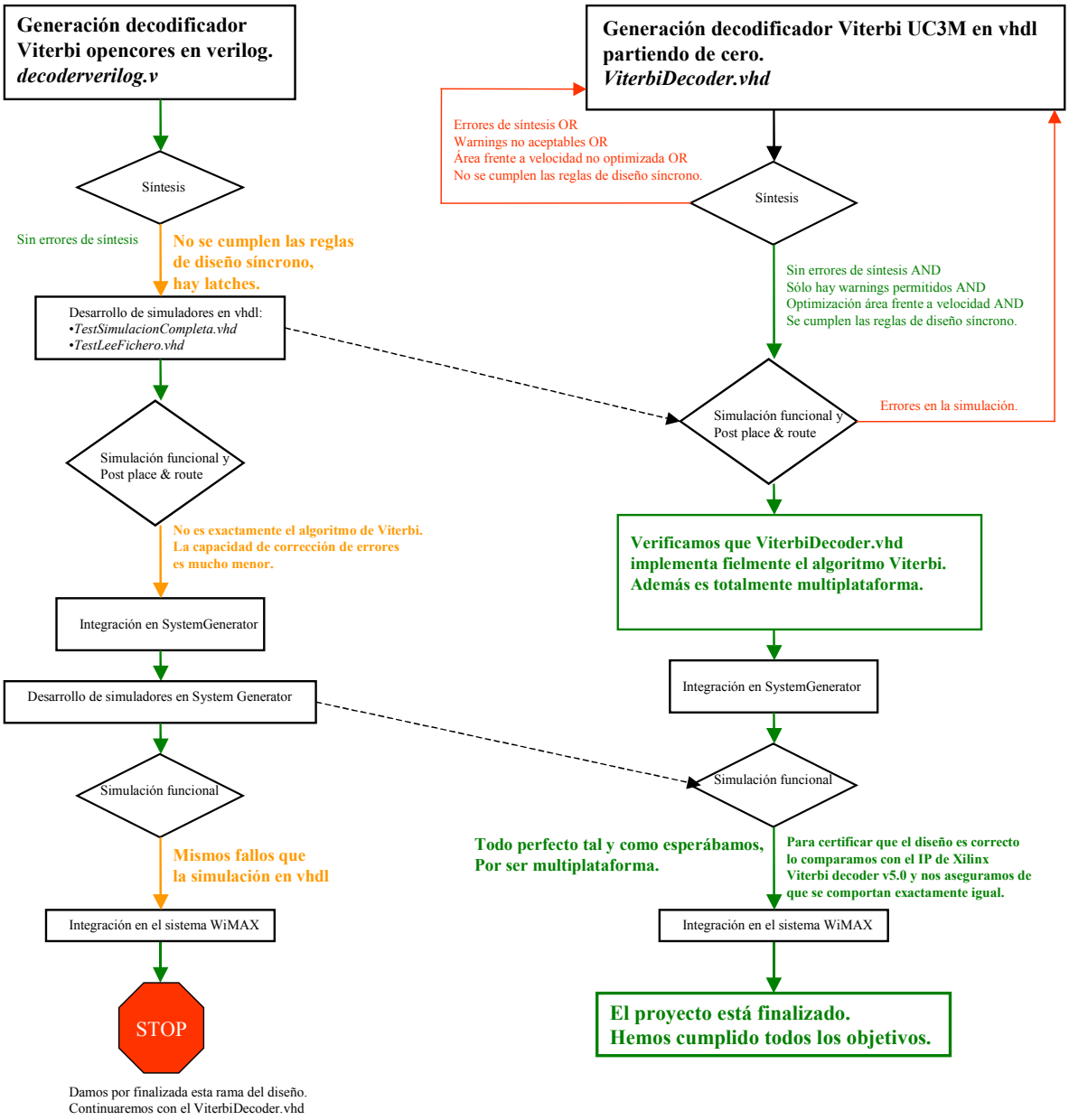


Figura 1.1: Pasos seguidos en el proyecto.

⟶ No se puede continuar sin corregir los errores.
⟶ El hito no está optimizado, pero funciona. Podemos continuar el trabajo y pasar al siguiente hito. Pero antes de finalizar el proyecto habrá que volver atrás y optimizarlo.
⟶ Hito totalmente finalizado y optimizado, no es necesario volver atrás.

En la figura anterior vemos que el decodificador de Opencores no es la mejor opción porque tiene latches. Además no es exactamente un decodificador Viterbi, y tiene una capacidad de corrección de errores muy inferior a la del auténtico algoritmo.

En el *capítulo 4* detallaremos los fallos del modelo de Opencores, marcados en naranja en la *figura 1.1*. En el *tema 7* lo compararemos con el decodificador UC3M, y demostraremos que el UC3M es capaz de corregir muchos más errores, porque sí que implementa exactamente el algoritmo Viterbi. El fallo de Opencores consiste en que divide la cadena de entrada en bloques independientes de 16 símbolos, y sobre cada uno de ellos aplica la decodificación Viterbi. Esto es incorrecto, porque el algoritmo Viterbi acepta tramas finitas en su entrada, pero debe decodificarlas de manera continua, en ningún caso puede dividir la cadena de entrada en bloques más pequeños.

1.4 Especificaciones.

Las características del prototipo WiMAX son:

- Estándar IEEE 802.16d-2004 (WiMAX Fijo). Especificación completa en [22].

- MIMO 2x2 OFDM. Multiplexado por división en frecuencias ortogonales en la capa física. Modulación de datos BPSK, (modulación binaria por salto de fase). Las siglas MIMO (*multiple input - multiple output*), indican que hay más de una antena transmisora y receptora. Lógicamente, 2x2 indica que hay 2 antenas en el transmisor y otras dos en el receptor.

Las características del decodificador Viterbi son:

- *Constraint length* K=7. Definida como la longitud en bits del polinomio generador. El codificador tiene K-1 = 6 registros de memoria. $2^{K-1} = 64$ estados.

- *Decode rate* R = ½, (tasa de decodificación = 1/2). 2 bits de entrada, 1 de salida.

- Polinomio generador 171, 133 (octal). 1111001 y 1011011 en binario.

- *Decoding depth* (profundidad de memoria) = 36.

- Decodificación *hard* (dura).

- *Trellis truncation* (truncamiento de trellis).

- 6 puertos de entrada.

- 2 puertos de salida.

- La especificación es la de un algoritmo Viterbi básico, por tanto las siguientes características no están incluidas en las especificaciones: *puncturing*, *zero tail* (cola de ceros), *tail biting* y decodificación *soft* (blanda). De manera que el decodificador que desarrollemos no debe tener ninguna de estas características.

En el *apartado 2.5.3* describimos las características anteriores, *nota 1.1*. Además al ser genéricas, se pueden consultar en cualquiera de las referencias del *apartado 2.8.1*.

Compararemos nuestro decodificador con el Xilinx IP core Viterbi decoder v5.0. Toda la documentación sobre este decodificador está en el *apartado 1.9.5.*

Nota 1.1: La característica puncturing no tiene interés para el proyecto. Por tanto no la describimos en esta memoria, pero el lector puede informarse sobre ella en: páginas 112-116 de la referencia [5] del *tema 2*; páginas 507-510 de [1] del *tema 2*. Hemos seleccionado estas dos referencias, pero se trata de una característica genérica, de manera que puede consultarse cualquiera de las citas del *apartado 2.8.1*.

El decodificador debe tener estas 6 entradas:

- clk: Reloj del sistema.
- $\overline{\text{aclr}}$: Reset asíncrono, activo a nivel bajo.
- ce: Clock enable (habilitación de la señal de reloj).
- EnableIn: Indica si hay datos disponibles en la entrada, activo a nivel alto.
- Din_1 : Corresponde a $Dout_1$ del codificador convolucional, polinomio 171.
- Din_0 : Corresponde a $Dout_0$ del codificador convolucional, polinomio 133.

Las 2 salidas deben ser:

- EnableOut: Indica si el bit de la salida es válido, activo a nivel alto.
- Data_Out: Bit de salida, que corresponde a la decodificación de un dato compuesto por Din_1 y Din_0.

La entrada del decodificador la constituyen símbolos compuestos por Din_1 y Din_0. Tras el tiempo de latencia, se obtiene la decodificación de cada símbolo en la salida. Este tiempo de latencia no está especificado, pero lógicamente debe ser lo menor posible.

La secuencia de datos en la entrada es continua. (Trama de longitud infinita).

Como la técnica es de truncamiento de trellis, en las tramas de entrada sólo habrá bits de datos, no es necesario incluir bits de relleno.

1.4.1 Medios hardware y software empleados.

Disponemos de una tarjeta de adquisición de datos analógicos: Lyrtech VHS-ADC Virtex 4. Toda la documentación necesaria sobre ella está en el *apartado 1.9.4*. Sus características más importantes son:

- FPGA integrada en la placa: Virtex 4 xc4vsx55 10ff1148.
- Memoria flash de 64 Mb para almacenar la programación de la FPGA.
- Programación de la FPGA mediante I^2C o JTAG.
- Tarjeta de adquisición de datos con 8 canales.
- Main Board VHS ADC 8.
- Memoria SDRAM de 128 MB con frecuencia 125 MHz.
- 8 ADCs (conversores analógico/digital). Con una capacidad regulable entre 1 y 105 megamuestras/segundo. Y una resolución de 14 bits por muestra (2 muestras por cada palabra de 32 bits). Además, la ganancia es independiente para cada canal y programable por software, siendo la entrada del tipo 50 Ω MMCX (coaxial micro miniaturizado).

En cuanto al software necesitamos 3 tipos de herramientas:

1. Xilinx ISE 8.1 service pack 3. Para realizar todo el flujo de diseño de la FPGA a partir del código VHDL que desarrollemos: síntesis, *map*, *place & route* y generación del archivo de configuración de la FPGA (*.bit). Con estas herramientas pasaremos del código VHDL a programar la FPGA, pasos 2, 3 y 4 del *apartado 1.3*.

2. Herramienta de simulación de hardware: Xilinx ISE Simulator. Para verificar que el código VHDL tiene el comportamiento que esperamos, antes de implementarlo en la FPGA. Pasos 5 y 6 del *apartado 1.3*.

3. System Generator. Consiste en una librería (*blockset*) de Simulink que modela bloques hardware de Xilinx. Utilizamos Matlab 7.1 release 14 service pack 3, que incluye Simulink. Con System Generator realizamos los pasos 7 al 10 de *1.3*.

La propia herramienta Matlab-Simulink-System Generator permite simular el diseño hardware realizado. Para ello desarrollamos un modelo de un sistema FEC, con la misma estructura que el simulador que hemos descrito con VHDL. Importaremos algunos módulos del simulador realizado con VHDL. Y también tendremos que codificar módulos auxiliares, utilizando funciones de Matlab y bloques de Simulink.

En el *apartado 1.9.2* está disponible una amplia bibliografía sobre System Generator. Que incluye tutoriales, guías de usuario, ejemplos, guías de referencia...

En las referencias [1], [2] y [3] del *tema 6*, está disponible la documentación básica sobre el manejo de Xilinx ISE 8.1 e ISE Simulator. En ellas se indica cómo realizar todo el proceso de diseño: síntesis, *map*, *place& route*, simulación funcional y post place & route, generación del archivo de configuración *.bit, paso del archivo *.bit a la placa... Además Xilinx dispone de una amplia documentación de referencia sobre todas sus herramientas, en la propia ayuda de los programas y en su página Web: http://www.xilinx.com

El manejo de las herramientas: Matlab-Simulink-System Generator, es obligatorio en la realización de este proyecto. Porque el prototipo WiMAX debe describirse con esta herramienta, ya que emplea bloques de la tarjeta Lyrtech que están modelados como bloques de System Generator.

Para conseguir el objetivo de que tanto el codificador como el decodificador sean multiplataforma, su descripción hardware debe realizarse mediante VHDL o Verilog. No puede realizarse instanciando bloques de System Generator, porque estos bloques dependen de la tecnología del fabricante Xilinx.

1.5 Estructura de la memoria.

Esta memoria se divide en 10 temas y 4 anexos.

- El anexo A contiene un listado con todos los archivos que hemos generado en el proyecto, incluyendo: código fuente VHDL, código fuente System Generator, simuladores, archivos de texto, código fuente Matlab y archivos de configuración (*.bit) de la FPGA Virtex 4 xc4vsx55 10ff1148. También indica los distintos diseños que hemos realizado y como están organizados en sus respectivas carpetas.

- Todos los archivos con código fuente y los simuladores están comentados con todo detalle. Estos comentarios se complementan con la descripción detallada que realizamos en esta memoria, en los capítulos 3, 4, 5 y 6. De manera que para comprender la arquitectura, manejo, interfaces y funcionalidad del código fuente y los simuladores, el lector tiene 3 opciones: los comentarios incluidos en el código, este documento, o ambas cosas.

- Al final de cada tema incluimos un apartado con las referencias que hemos empleado como base para realizar tanto el código como la memoria. También incluimos las fuentes bibliográficas donde se pueden ampliar conocimientos. En algunas ocasiones las referencias son comunes para varios temas. Para no repetirlas, desde un capítulo se puede referenciar bibliografía de otros temas. También incluimos en el CD del proyecto una copia de toda la documentación disponible sin derechos de autor (*copyright*).

- Hemos desarrollado dos decodificadores Viterbi:

    - *ViterbiDecoder.vhd*: Es un desarrollo completamente propio. Todo el proceso de diseño e implementación lo realizamos completamente nosotros, sin tomar como base ni el modelo de Opencores, ni ningún otro decodificador. Por eso también lo denominamos decodificador UC3M. En los temas 5 y 7 demostramos que *ViterbiDecoder.vhd* cumple exactamente el algoritmo Viterbi y no tiene fallos funcionales, ni de diseño, ni implementación. Además lo optimizamos tanto en área como en velocidad. De manera que este bloque es el módulo final de nuestro proyecto y cumple todas las especificaciones.

    - *decoderverilog.v*: Lo desarrollamos basándonos en un modelo de la Web de Opencores. En los temas 4 y 7 demostramos que *decoderverilog.v* tiene errores y no realiza exactamente la función de un algoritmo Viterbi. Además tiene algunos fallos de diseño. Así que el bloque está sin terminar, pero no continuamos su desarrollo porque únicamente lo hemos utilizado como modelo intermedio. No tiene sentido continuar con su desarrollo para optimizarlo y corregir sus errores, porque el objetivo del proyecto es trabajar con nuestro propio decodificador, en este caso el *ViterbiDecoder.vhd*.

- En este tema 1 indicamos: los objetivos de nuestro proyecto, los pasos básicos que hemos dado para poder realizarlo, las especificaciones y los recursos disponibles. También documentamos los conocimientos básicos que son necesarios para el proyecto: VHDL, diseño multiplataforma y flujo de diseño en una FPGA.

- El tema 2 es una descripción teórica de: la codificación convolucional, decodificación Viterbi y sistemas FEC. En este capítulo no nos centramos en nuestro desarrollo particular, sino que realizamos una descripción teórica de los conocimientos básicos que se necesitan para realizar y comprender el proyecto.

- En los temas 3, 4, 5 y 6 describimos detalladamente los diseños que hemos realizado en el proyecto. Documentamos todos los aspectos de interés, centrándonos en estos puntos: características, interfaz entrada-salida, arquitectura, funcionamiento y manual de usuario. Siempre describimos todos los bloques, tanto los principales como los de jerarquía inferior.

    - En el tema 3 se documenta el codificador convolucional que hemos implementado: *EncoderK7.vhd*.

    - El tema 4 documenta el decodificador Viterbi que hemos implementado partiendo del modelo de Opencores: *decoderverilog.v*.

    - El capítulo 5 describe el decodificador Viterbi definitivo que constituye el objetivo final de este proyecto: *ViterbiDecoder.vhd*.

    - El capítulo 6 describe los simuladores que hemos realizado. Los cuales nos permiten verificar que el codificador convolucional y el *ViterbiDecoder.vhd* son correctos y cumplen las especificaciones. También nos sirven para detectar que *decoderverilog.v* no cumple con las especificaciones, y por tanto no es un decodificador Viterbi funcionalmente correcto.

- En el tema 7 recopilamos todos los resultados que se obtienen tras realizar la síntesis y simulación de todo el código que hemos implementado en el proyecto. Estos resultados determinan: el área ocupada, la frecuencia máxima de reloj y el rendimiento, (medido en gráficas BER frente a Eb/No). Además comparamos los diferentes diseños entre sí. Por último, comparamos todos los diseños que hemos realizado con el decodificador Viterbi que utilizamos como referencia: el Xilinx IP core Viterbi decoder v5.0.

- En el tema 8 integramos el bloque *ViterbiDecoder.vhd* en el prototipo WiMAX que estamos realizando entre varios proyectos fin de carrera diferentes.

- En el capítulo 9 calculamos los costes del proyecto y elaboramos un presupuesto. Además comparamos el presupuesto con el precio de mercado de un decodificador Viterbi comercial. (El Xilinx IP core Viterbi decoder).

- Por último, en el tema 10 recopilamos todas las conclusiones finales de nuestro trabajo. También proponemos trabajos futuros que pueden realizarse tomando como base nuestro proyecto finalizado.

- En los anexos B y C mostramos algunos desarrollos matemáticos, correspondientes al tema 2, que no hemos incluido en el tema 2 para no hacerlo demasiado extenso.

- Por último, en el anexo D recopilamos los resultados tras realizar la síntesis y el map, de todos los archivos con código fuente VHDL que hemos implementado.

1.6 VHDL.

VHDL es un lenguaje de descripción hardware, HDL (Hardware Description Language). Son las siglas de VHSIC HDL, Very High Speed Integrated Circuit Hardware Description Languaje.

El lenguaje se utiliza para el diseño, verificación y documentación de sistemas digitales. Mediante el código VHDL se describe la arquitectura y funcionalidad de un circuito hardware, y también se utiliza para simular el circuito. A continuación, las herramientas de síntesis desarrolladas por los fabricantes sintetizan el código en el hardware.

VHDL es un lenguaje estándar usado para describir sistemas digitales, norma IEEE 1076. Y se trata junto a Verilog, del HDL de referencia soportado por la gran mayoría de herramientas de síntesis y de dispositivos lógicos programables. Otro lenguaje HDL importante es ABEL, (Advanced Boolean Equation Language). Pero ABEL no se utiliza tanto, ni es soportado por tantas herramientas de síntesis como VHDL y Verilog.

Se estandariza por primera vez en 1987, norma IEEE 1076-1987, "IEEE Standard VHDL Language Reference Manual". En 1993 se le añaden algunas mejoras, resultando la norma IEEE 1076-1993. Esta es la referencia del VHDL, la más utilizada de todas y la que empleamos para todo el código desarrollado en este proyecto.

A continuación, como en todas las normas IEEE se somete a revisiones periódicas en 2000, 2002 y 2008. En estas revisiones se hacen algunos añadidos respecto a la norma anterior, pero la norma estándar de referencia sigue siendo IEEE 1076-1993.

En el *apartado 1.9.1* está la bibliografía con la especificación completa de VHDL, ejemplos, tutoriales y manuales de uso.

1.7 Diseño hardware multiplataforma.

Ya hemos indicado, que uno de los objetivos tanto del decodificador Viterbi, como del codificador convolucional y los simuladores codificados con VHDL, es que sean multiplataforma.

Para conseguir que un diseño sea multiplataforma, el código debe estar desarrollado completamente con VHDL o Verilog. De esta manera el hardware que se describa con este código, podrá implementarse con cualquier herramienta de síntesis y en cualquier soporte hardware que acepte VHDL o Verilog. Como hemos visto, estos dos lenguajes HDL son estándares en la industria, de manera que la gran mayoría de fabricantes y de dispositivos lógicos programables los aceptan.

El diseño hardware deja de ser multiplataforma cuando se utiliza un core proporcionado por el fabricante. Un core es un módulo implementado por el fabricante utilizando su propia tecnología. Está optimizado para usarse en determinados dispositivos de su propio catálogo de productos, y con determinadas herramientas de síntesis.

Estos cores implementan diversas funciones, como: contadores, decodificadores, memorias, filtros, funciones matemáticas, moduladores... ver *figuras 1.2 y 1.3*. Cada

fabricante los nombra de una manera diferente: IP (Coregen & Architecture Wizard) o LogiCore es la denominación de Xilinx, (ver *figura 1.2*). Por simplicidad, en este documento utilizaremos simplemente la denominación IP Core.

Algunos IP cores no son gratuitos y se necesita una licencia para poder sintetizarlos en hardware, por ejemplo el Viterbi decoder. Sin embargo, otros son gratuitos y pueden sintetizarse libremente en el hardware, por ejemplo: codificador convolucional, memorias, contadores, sumadores, comparadores...

En los diseños es habitual emplear IP cores porque acortan el tiempo de desarrollo. Debido a que nos proporcionan módulos ya codificados que no tendremos que desarrollar nosotros mismos. Sin embargo, en este proyecto no podemos usarlos, porque los IP cores dependen tanto del hardware como de la herramienta de síntesis:

a) Un IP core de un fabricante no puede sintetizarse en hardware de otro fabricante.

b) Además, un IP core no puede utilizarse en todos los dispositivos de su mismo fabricante. Por ejemplo, puede ocurrir que realicemos un diseño optimizado sobre una tarjeta. Pero si en el futuro se cambian esas tarjetas por otras más modernas del mismo fabricante, en algunas ocasiones los cores no valdrán y habrá que actualizarlos.

c) Por último, aún sintetizando sobre el mismo hardware, un core no siempre puede trasladarse de una herramienta de sínteis a otra. Incluso en el caso de que ambas sean del mismo fabricante. Por ejemplo, si utilizamos Xilinx ISE 8.1 e instanciamos cores sobre la Virtex4 xc4vsx55, puede que estos cores no nos valgan si pasamos por ejemplo al Xilinx ISE 12.1, aunque mantengamos la misma FPGA.

Debido a estos inconvenientes, en nuestro diseño nunca utilizaremos IP cores, sino que seremos nosotros mismos los que codifiquemos todos los módulos que sean necesarios. Esto supone una dificultad añadida al diseño, pero es la única forma de garantizar que será totalmente multiplataforma.

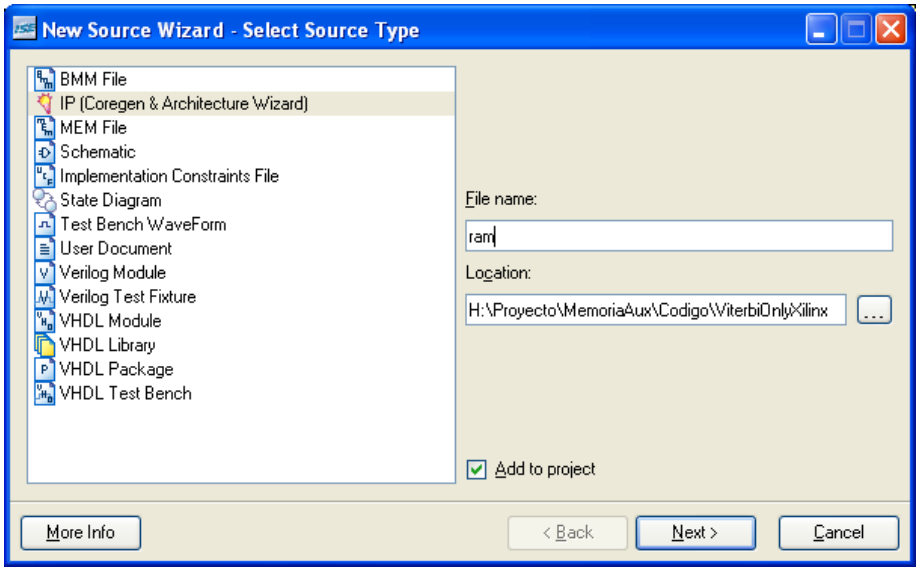


Figura 1.2: Creación de un IP core en Xilinx ISE8.1.

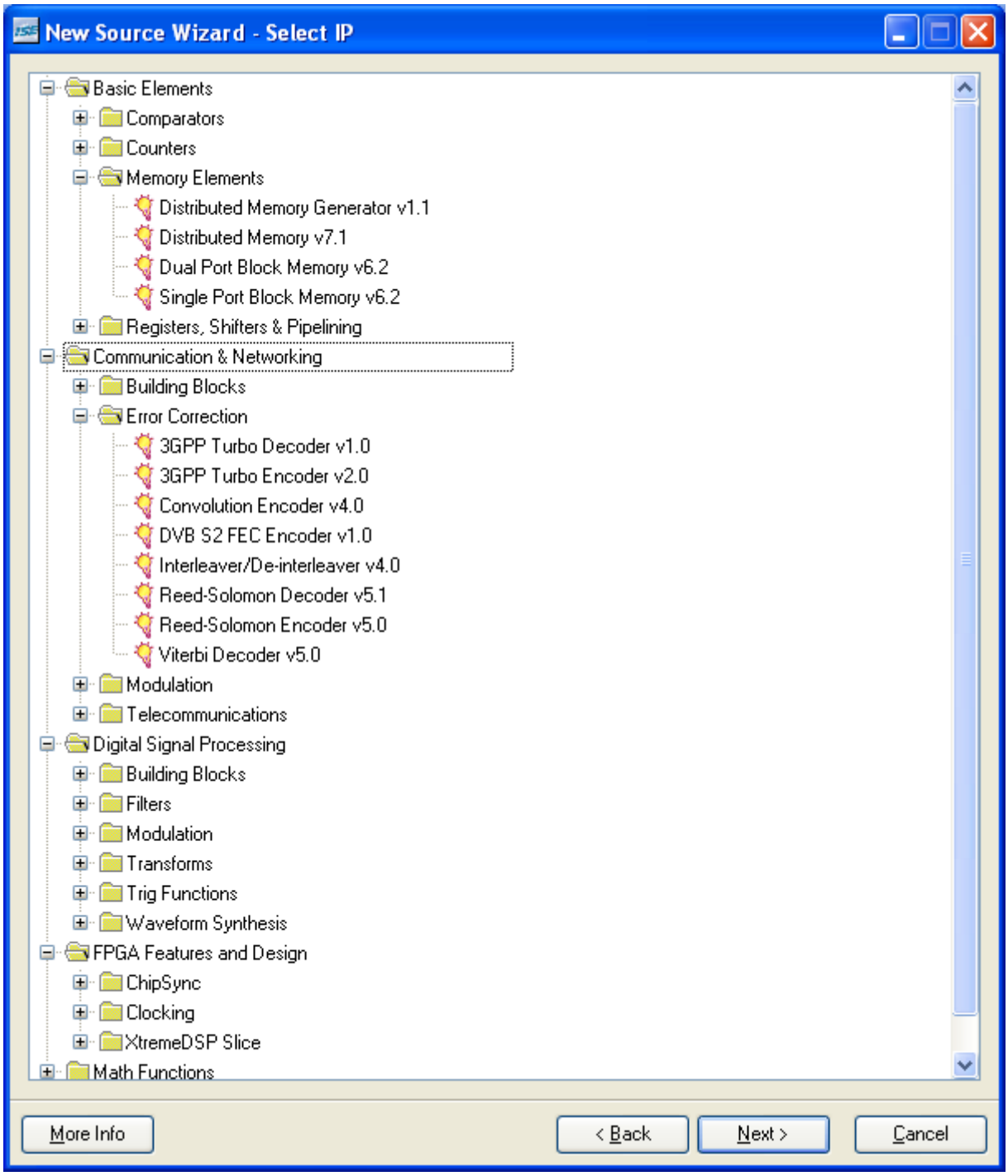


Figura 1.3: Ejemplos de IP cores disponibles en Xilinx ISE 8.1.

1.8 Flujo de diseño en una FPGA.

El objetivo final del proyecto es obtener un código fuente en VHDL que implemente el decodificador Viterbi. Para ello hay que seguir los pasos que aparecen en la siguiente figura:

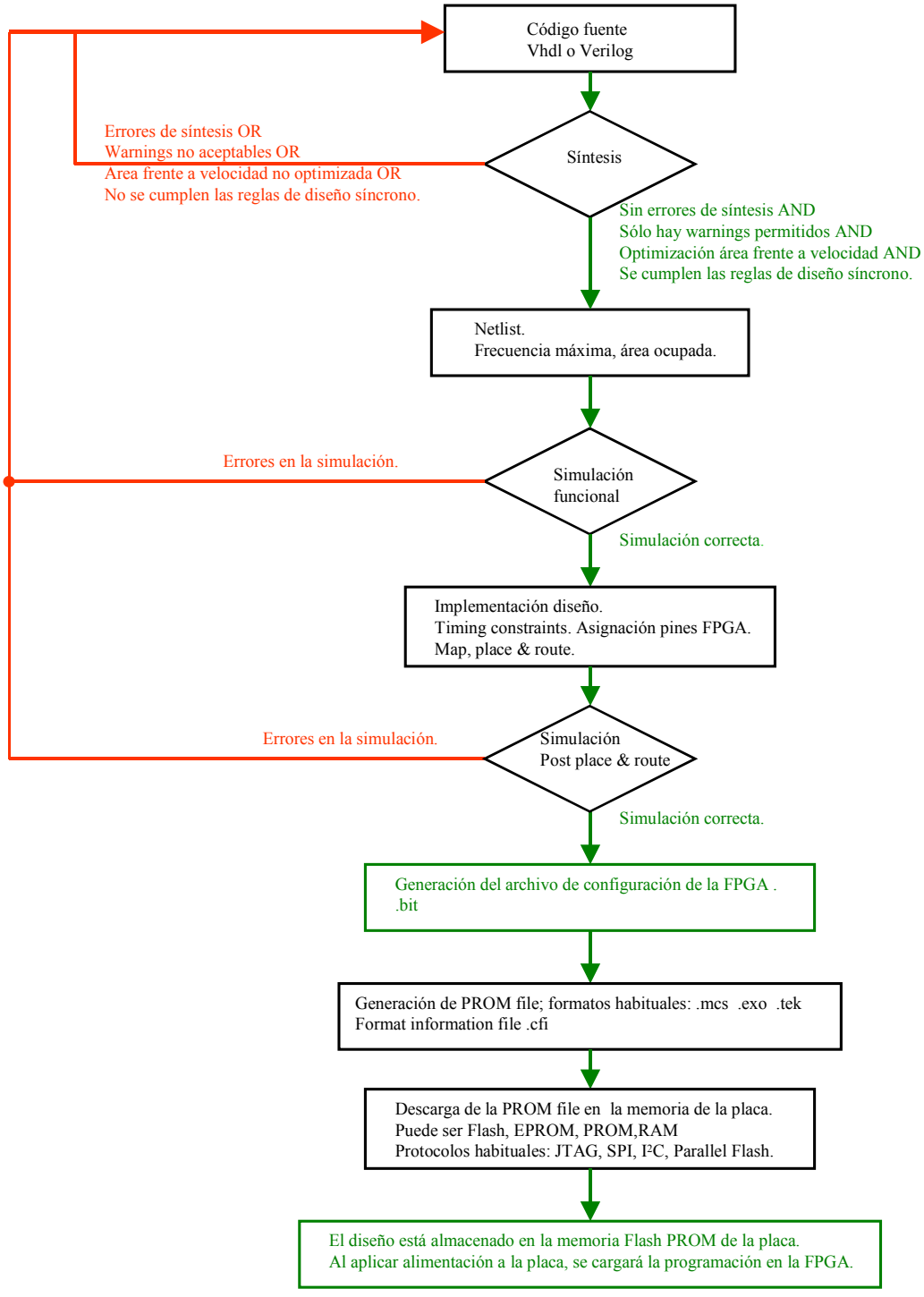


Figura 1.4: Flujo de diseño en una FPGA.

Debemos cumplir las reglas de diseño síncrono, que consultamos en [63], [64] y son:

1. No hay latches, [65] y [66].

2. No hay realimentaciones en la lógica combinacional.

3. Todos los circuitos secuenciales deben realizarse utilizando *flip-flops*, preferentemente de tipo D, sensibles al mismo flanco de reloj.

4. El circuito debe disponer de una señal de reloj única y común para todos los *flip-flops* del circuito.

5. A todos los flip-flops les llega simultáneamente la señal de reloj del circuito. Para conseguir esto el reloj debe llegar directamente a todos los pines clk de los flip-flops. No puede haber nada, ni lógica ni flip-flops, entre la entrada clk de un bloque y los pines clk de los flip-flops del bloque.

6. Las entradas de reset asíncronas de los flip-flops del circuito sólo pueden conectarse a la señal de reset de entrada al circuito. Esto significa que la activación/desactivación del reset la controla siempre una señal externa. En ningún caso el propio circuito puede controlar la activación/desactivación del reset asíncrono de los flip-flops.

7. No puede haber lógica combinacional entre la entrada de reset de un bloque y los pines de reset de los flip-flops del bloque.

8. Todas las entradas a los circuitos combinacionales están registradas.

Una vez sintetizado el código sin errores, debemos analizar los warnings que aparecen al sintetizar, para asegurarnos de que ninguno de ellos afecta a la arquitectura ni al funcionamiento.

En todo diseño hardware es fundamental optimizar el código, disminuyendo el área ocupada y aumentando la frecuencia máxima de reloj.

Una vez realizados todos estos pasos, se procede a realizar la simulación funcional del circuito. Debe ser exhaustiva, para asegurarnos de que probamos el funcionamiento del circuito en todas las situaciones que puedan darse en el sistema real. Para ello es necesario modelar lo más fielmente posible el entorno real que tendrá el sistema cuando se implemente en el hardware.

El simulador se realiza definiendo todas las señales externas que pueden entrar al circuito. Se utilizan vectores de test que cubran todas las combinaciones posibles de las señales de entrada. Esto a veces no es viable y no se puede abarcar el 100 % de las combinaciones en la entrada.

La simulación funcional nos garantiza que el circuito se va a comportar en la placa hardware real igual que en la simulación, siempre que no se supere la frecuencia máxima de reloj que obtiene el sintetizador. Pero no se puede garantizar con certeza

absoluta, para ello la única manera es probarlo en la placa. A este proceso de prueba sobre el dispositivo hardware real se le denomina emulación.

El siguiente paso es la implementación, asignamos los pines de entrada y salida de la FPGA y ponemos las restricciones de tiempo. A continuación realizamos el *map* y el *place & route*.

Es conveniente realizar una simulación post place & route a pesar de haber realizado ya la funcional. El código del simulador es el mismo, los vectores de test de entrada y los resultados esperados también. Así que realizar esta simulación no requiere coste de desarrollo. El único problema es que el tiempo de computación es más alto.

La simulación funcional utiliza un modelo ideal del hardware, en el que los retardos de la lógica combinacional, tiempo de propagación, set up y hold de los flip-flops, skew... son cero. En la post place & route se utiliza un modelo real del hardware, en el que se incluyen todos los tiempos y retardos, por tanto es más precisa.

Tras la simulación post place & route, se garantiza que el circuito se comportará en la placa hardware igual que en la simulación, con una probabilidad de prácticamente el 100 %. Aunque no se puede garantizar con certeza absoluta, para ello la única forma es emularlo sobre el hardware.

Realizar la simulación post place & route tiene estas 2 ventajas:

1. Es más exacta que la funcional, por lo que la probabilidad de que el circuito se comporte en la placa igual que en la simulación es mayor, prácticamente del 100%.

2. La funcional garantiza un funcionamiento por debajo de la frecuencia máxima de reloj que obtiene el sintetizador. Pero esa frecuencia no es exacta, porque está calculada antes de interconectar los pines y los bloques del circuito. En cambio, tras el place & route se obtiene una frecuencia máxima más precisa.

La desventaja es que debido a su mayor precisión el coste computacional es más alto y dependiendo del tamaño del diseño, puede llevar mucho tiempo ejecutar la simulación.

Una vez finalizadas las simulaciones, ya se conoce como será el funcionamiento del circuito en el hardware real. De manera que lo único que falta es implementar el código VHDL en el hardware. Esta tarea es sencilla y directa, no requiere ninguna realimentación que obligue a modificar el código VHDL que describe el circuito, de manera que este código es el definitivo. Esto se aprecia claramente en la *figura 1.4*, donde tras acabar la simulación se sigue siempre un camino verde. No se produce ninguna situación que obligue a seguir un camino rojo, que nos realimentase a hitos anteriores.

El proceso de implementar el código VHDL en el hardware depende de las herramientas de síntesis y el hardware que se vaya a emplear. Pero en todos los casos se trata de unos pasos definidos y sin ninguna complicación. Lo habitual es que la placa tenga una memoria para almacenar la programación. En ese caso, se genera un archivo de configuración *.bit, después una PROM file y se descarga la PROM file en la memoria.

El resultado final es que la programación queda almacenada en la memoria de la placa. Si es no volátil, cada vez que se alimente la tarjeta, la FPGA se programará automáticamente cargando el contenido de la memoria Flash/EPROM/PROM. En caso de que sea RAM, la programación se pierde, por lo que cada vez que se encienda la tarjeta habrá que descargarle la programación en la RAM.

Para ampliar los conocimientos sobre el flujo de diseño en una FPGA disponemos de las referencias [67], [68] y [69].

1.8.1 Pasos para implementar el diseño en un hardware diferente.

En todo el proceso de diseño utilizamos unas herramientas de síntesis y una FPGA concretas. Nosotros usamos la FPGA que tenemos en el laboratorio: Virtex 4 xc4vsx55 10ff1148 y las herramientas Xilinx ISE 8.1.

Pero ya hemos indicado que el código VHDL que hemos desarrollado es independiente de la tecnología, por tanto no sólo funcionará sobre la FPGA que hemos utilizado en este proyecto. Funcionará sobre cualquier otro dispositivo hardware programable y de cualquier fabricante, siempre que acepte VHDL.

Para implementar el código fuente VHDL que hemos desarrollado, en un hardware diferente, habría que utilizar las herramientas de síntesis que proporcione el fabricante del nuevo hardware. Partiríamos del código fuente VHDL inicial, a continuación lo sintetizaríamos, haríamos el map, el place & route, obtendríamos el archivo de configuración *.bit y por último, descargaríamos el archivo de configuración en el nuevo hardware.

El proceso consistiría únicamente en seguir las flechas verdes de la *figura 1.4*. Se trata de un proceso automático, no hace falta hacer una realimentación, siguiendo las flechas rojas para hacer cambios en el código VHDL del que partimos. Porque en los diversos hitos del proceso siempre se dará la condición de seguir un camino verde, nunca debe suceder la condición que obligue a seguir un camino rojo. Para ello tuvimos que cumplir los siguientes requisitos cuando diseñamos el código VHDL inicial:

1. Hemos verificado que el código se sintetiza y funciona correctamente sobre una FPGA concreta.

2. Utilizamos solamente VHDL, sin IP cores.

3. Cumplimos todas las reglas de diseño síncrono y también todas las reglas indicadas en la *figura 1.4*.

Cumpliendo todos los requisitos anteriores, el diseño podrá funcionar sobre cualquier hardware, sin realizar ningún cambio en el código fuente VHDL inicial. Lo único que cambiará es la frecuencia máxima de reloj y el área ocupada. Porque estos parámetros dependen del dispositivo hardware que empleemos.

1.9 Referencias

Proporcionamos el enlace Web siempre que exista, accedemos a estos enlaces por última vez en noviembre de 2011. Además tenemos una copia de seguridad en el CD del proyecto de toda la documentación sin copyright.

1.9.1 VHDL.

[1] Norma IEEE Std 1076™ –2002. "IEEE Standard VHDL Languaje Reference Manual". Aprobado el 26 de Julio de 2002 por ANSI, American National Standards Institute; y el 21 de Marzo de 2002 por IEEE-SA Standards Board.
http://vhdl-manual.narod.ru/books/ieee_manual.pdf
http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4772738

[2] Douglas L.Perry. "VHDL Programming by Example". McGraw-Hill, 2002.

[3] Miguel Ángel Freire Rubio. "Introducción al lenguaje VHDL". Universidad Politécnica de Madrid, EUITT, Escuela Universitaria de Ingeniería Técnica de Telecomunicación, Departamento de Sistemas electrónicos y de Control, 1999.
http://www1.unex.es/eweb/fisteor/antonio_astillero/ec/vhdl/Manual%20VHDL.pdf

[4] Synario Design Automation. "VHDL Reference Manual". Marzo 1997.
http://www.usna.edu/EE/ee462/manuals/vhdl_ref.pdf

[5] Pong P. Chu. "RTL Hardware Design Using VHDL. Coding for Efficiency, Portability, and Scalability". Wiley-Interscience. 2006.

[6] Jan Van der Spiegel. "VHDL Tutorial"
http://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html

[7] Documento Xilinx: "VHDL Reference Guide". 1999.
http://een.iust.ac.ir/profs/mirzakuchaki/vhdl/tutorial/VHDL-xilinx-help.PDF

[8] "VeriBest FPGA Synthesis VHDL Reference Manual".
http://courses.cit.cornell.edu/ee475/tutorial/VHDLman.pdf

[9] Ben Cohen. "VHDL Coding Styles and Methodologies... an In-Depth Tutorial". Kluwer Academic Publishers, 1995.

[10] David Naylor. "VHDL: A Logic Synthesis Approach". Chapman & Hall, 1997.

1.9.2 System Generator.

Al realizar el proyecto disponemos de la versión 8.1. La documentación de esta versión no está disponible vía Web en el momento de redactar este documento, octubre 2011. Por eso referenciamos la v13.3, que es la disponible en el momento de la redacción.

La documentación de la versión 8.1 siempre estará disponible en la ayuda del programa. Pero esa ayuda no la podemos incluir aquí como referencia porque no está en la Web. Pero sí la incluimos en la documentación que entregamos al realizar el CD del proyecto.

[11] Sitio Web herramienta System Generator de Xilinx:
http://www.xilinx.com/tools/sysgen.htm

[12] Documento Xilinx: "System Generator for DSP. Reference Guide". ug638, v13.3, 19 Octubre 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sysgen_ref.pdf

[13] Documento Xilinx: "System Generator for DSP. Getting Started Guide". ug639, v13.3, 19 Octubre 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sysgen_gs.pdf

[14] Documento Xilinx: "System Generator for DSP. User Guide". ug640, v13.3, 19 Octubre 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sysgen_user.pdf

[15] Sistemas operativos compatibles con System Generator:
http://www.xilinx.com/ise/ossupport/index.htm

[16] Versiones de Matlab compatibles con System Generator:
http://www.xilinx.com/support/answers/17966.htm

[17] Sitio Web: "MathWorks España – Matlab and Simulink for Technical Computing".
http://www.mathworks.es/index.html

[18] Sitio Web con acceso a toda la documentación de MathWorks:
http://www.mathworks.es/help/?s_cid=global_nav

[19] Justin Delva, Adrian Chirila-Rus and Shay Seng. "Using System Generator for Systematic HDL Design, Verification, and Validation". White Paper: Xilinx System Generator. wp283, v1.0, 17 enero 2008.
http://www.xilinx.com/support/documentation/white_papers/wp283.pdf

Desde la Web de MathWorks se puede conseguir toda la documentación de Matlab y Simulink. A continuación ponemos un ejemplo, le referencia [20]. No incluimos más referencias porque la mejor forma de acceder a ellas es directamente desde la página de MathWorks. Además se requiere registro, por lo que no podemos citar en este documento las referencias más importantes.

[20] Documentación MathWorks: "Integrating Xilinx System Generator and Simulink HDL Coder". Application Guidelines. Versión 1.0 de Febrero de 2008.
http://www.mathworks.com/tagteam/59570_91561v01_XilinxHDL_AppGuidelines_FINAL.pdf

[21] Documento Xilinx System Generator for DSP. "ML506 DSP Hardware Co-Simulation with Xilinx System Generator for DSP 10.1i SP2", August 2008.
http://www.xilinx.com/products/boards/ml506/ml506_10.1_2/docs/ml506_sysgen_dsp_hw_cosim_creation.pdf

1.9.3 WiMAX.

[22] Norma IEEE Std 802.16™ –2004. "IEEE Standard for Local and metropolitan area networks. Part 16: Air Interface for Fixed Broadband Wireless Access Systems". 1 Octubre 2004.
http://wirelessafrica.meraka.org.za/wiki/images/8/83/802.16-2004.pdf

[23] David Díaz Martín. "Prototipado de un Sistema WiMAX MIMO 2x2". Proyecto Fin de Carrera en Ingeniería Técnica Superior de Telecomunicación. Universidad Carlos III de Madrid. Enero 2010.

[24] David Díaz Martín y Roberto Prieto Alonso. "Estudio Práctico sobre el prototipado de un sistema MIMO 2x2 para WiMAX" . Estudio Tecnológico en Ingeniería Técnica Superior de Telecomunicación. Universidad Carlos III de Madrid. 24 Enero 2008.

[25] Frank Ohrtman. "WiMAX Handbook. Building 802-16 Wireless Networks. Complete annotation of 802.16 specification. WiMax Quality of Service (QoS). WiMax security. WiMax and WiFi". McGraw-Hill Communications. 2005.

[26] Jeffrey G. Andrews; Arunabha Gosh and Rias Muhamed. "Fundamentals of WiMAX. Understanding Broadband Wireless Networking".Prentice Hall Communications Engineering and Emerging Technologies Series. February 2007.

[27] Francisco Javier López Martínez. "Diseño de Transmisor y Receptor para Redes Inalámbricas y W-MAN". Proyecto Fin de Carrera en Ingeniería Técnica Superior de Telecomunicación. Universidad de Málaga. 7 Julio 2005.

[28] José Ángel Rivas Cantero. "Estimación de Canal y Sincronización de Frecuencia en un Sistema MIMO-OFDM Compatible con el Estándar WiMAX IEEE 802.16-2004". Proyecto Fin de Carrera en Ingeniería Técnica Superior de Telecomunicación. Universidad Carlos III de Madrid. Junio 2006.

[29] Tomás Alemany Sanchez. "Descripción Hardware de Algoritmos de estimación de canal y sincronización tiempo-frecuencia para un sistema 2x2 MIMO-OFDM". Proyecto Fin de Carrera en Ingeniería Técnica Superior de Telecomunicación. Universidad Carlos III de Madrid. Octubre 2010.

1.9.4 Tarjeta Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 xc4vsx55.

[30] Sitio Web Lyrtech
http://www.lyrtech.com/

[31] Sitio Web tarjeta Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 xc4vsx55 10ff1148
http://www.lyrtech.com/products/vhs-adc.php#

[32] Specification Sheet tarjeta Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 xc4vsx55 10ff1148:
http://www.lyrtech.com/documents/product_sheets/Specification%20sheet%20-%20VHS-ADC%20%28hi_res%29.pdf

El Specification Sheet es accesible vía Web, pero no lo son el data sheet, y las guías de usuario. Esta documentación hay que pedírsela a Lyrtech para que la suministre, por eso no podemos referenciarla en este documento. Pero nosotros disponemos de ella y la entregamos en el CD del proyecto.

Además de las guías de Lyrtech, a continuación citamos documentación de Xilinx relativa a la Virtex-4 XC4VSX55 10FF1148:

[33] Página Web de la Xilinx Virtex-4:
http://www.xilinx.com/support/index.htm#nav=sd-nav-link-19224&tab=tab-sd

[34] Xilinx "Virtex-4 Family Overview". ds112.pdf, v3.1, 30 de Agosto 2010:
http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[35] Xilinx "Virtex-4 FPGA User Guide". ug070.pdf, v2.6, 1 de Diciembre de 2008:
http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[36] Xilinx "Virtex-4 FPGA Configuration User Guide". ug071, v1.11, 9 Junio 2009:
http://www.xilinx.com/support/documentation/user_guides/ug071.pdf

[37] Xilinx "Virtex-4 FPGA Packaging and Pinout Specification". ug075, v3.3, 19 Septiembre 2009:
http://www.xilinx.com/support/documentation/user_guides/ug075.pdf

[38] Xilinx "Virtex-4 FPGA Data Sheet: DC and Switching Characteristics". ds302, v3.7, 9 septiembre 2009.
http://www.xilinx.com/support/documentation/data_sheets/ds302.pdf

[39] Xilinx "Virtex-4 Family Package/Device Pinout Files (ASCII)":
http://www.xilinx.com/support/packagefiles/virtex-4-pkgs.htm

[40] Pinout encapsulado 4vsx55ff1148 de Xilinx, 10 Junio 2005:
http://www.xilinx.com/support/packagefiles/v4packages/4vsx55ff1148.txt

1.9.5 Xilinx LogiCore IP Viterbi decoder v7.0

Nota 1.2: La documentación que referenciamos corresponde a la versión Viterbi decoder 7.0 y en nuestro código usamos la 5.0. Nos vemos obligados a utilizar la 5.0 porque es la versión disponible en el software del que disponemos: System Generator v8.1. Sin embargo, en el momento de redactar este documento, octubre 2011, no estaba disponible en la Web la documentación del 5.0, sólo estaba accesible vía Web la documentación del 7.0. Pero no supone ningún problema, porque la documentación del 5.0 siempre estará disponible en la ayuda de System Generator. No la referenciamos porque la ayuda no es accesible vía Web. Pero sí la incluimos en la documentación que entregamos al realizar el CD del proyecto.

[41] Página Web de Xilinx: http://www.xilinx.com/

[42] Página Web decodificador Viterbi Xilinx:
http://www.xilinx.com/products/intellectual-property/Viterbi_Decoder.htm

[43] Xilinx "Viterbi Decoder v7.0 Data Sheet". ds247.pdf, 1 Marzo 2011:
http://www.xilinx.com/support/documentation/ip_documentation/viterbi_ds247.pdf

[44] Xilinx "LogiCORE IP Viterbi Decoder v7.0 User Guide". ug745.pdf, v1.0, 18 Octubre 2010:
http://www.xilinx.com/support/documentation/ip_documentation/ug745_viterbi_decoder.pdf

[45] Documento Xilinx: "System Generator for DSP. Reference Guide". UG638, v13.3, October 19, 2011. Páginas 413-417.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sysgen_ref.pdf

[46] Distribuidor de Xilinx en España, empresa Silica: Gabriel Cutillas, Field Application Engineer. Gabriel.Cutillas@silica.com
http://www.silica.com

[47] Xilinx "Viterbi Synchronization". ds205, v1.0, 30 Abril 2002.
http://www.xilinx.com/ipcenter/catalog/logicore/docs/viterbi_synchronization.pdf

[48] Michael Francis. "Viterbi Decoder Block Decoding – Trellis Termination and Tail Biting". Documentación de Xilinx, XAPP551 (v2.0), 30 de Julio de 2010.
http://www.xilinx.com/support/documentation/application_notes/xapp551.pdf

[49] Documento xilinx: "Xilinx INC. Core License Agreement", enero 2009:
http://www.xilinx.com/ipcenter/doc/xilinx_click_core_site_license.pdf

[50] Xilinx, "Question and Answers. SignOnce IP License Agreement", 9 Agosto 2007:
http://www.xilinx.com/ipcenter/CLA_QA.pdf

1.9.6 Referencias genéricas.

[51] Christian B. Schlegel and Lance C. Pérez. "Trellis and Turbo Coding". IEEE Press Series on Digital & Mobile Communication, pp. 3-10, 2004.

[52] Jorge Castiñeira Moreira and Patrick Guy Farrell. "Essentials of Error-Control Coding". Wiley, pp 41, 42; 65-76, 2006.

[53] Chip Fleming. "A Tutorial on Convolutional Coding with Viterbi Decoding". Spectrum Applications 11 Febrero de 2006.
http://pw1.netcom.com/~chip.f/viterbi/tutorial.html
cfleming@ieee.org

[54] Herbert Dawid, Olaf J. Joeressen and Heinrich Meyr. "Chapter 17 Viterbi Decoders: High Performance Algorithms and Architectures". Páginas 1-3.
http://www.eecs.berkeley.edu/newton/Classes/EE290sp99/lectures/ee290aSp996_1/vit_chap17.pdf

[55] IP Data Sheet Block Viterbi Decoder. Lattice Semiconductor Corporation. Octubre 2004. Página 1. http://www.latticesemi.com/lit/docs/ip/ip1033.pdf

[56] Viterbi Decoder Datasheet. Seasolve Software Inc, 2008. Página 1.
http://www.seasolve.com/wireless-ip-brochures/viterbi-decoder.pdf

[57] Louis Litwin; Michael Pugel; Rod Rhodes and John Richardson. "ADSL Technology Explained, Part1: The Physical Layer". EE Times Design, 1 Marzo 2001.
http://www.eetimes.com/design/communications-design/4140019/ADSL-Technology-Explained-Part-1-The-Physical-Layer

[58] Thomas Starr; Massimo Sorbara; John M. Cioffi and Peter J. Silverman. "DSL Advances". Prentice Hall Communications Engineering and Emerging Technologies series. Páginas 47-50; 162-166, 5 Enero 2003.

[59] Andres Iborra y Juan Suardiaz. "Unidad 4 Dispositivos Lógicos Programables". Universidad Politécnica de Cartagena, Febrero 2003.
http://www.dte.upct.es/personal/andres_iborra/docencia/sis_elec/pdfs/t4.pdf

[60] Uwe Meyer-Baese. "Digital Signal Processing with Field Programmable Gate Arrays". Springer, pp. 3-27, 2001.

[61] Stephen Brown and Zvonko Vranesic. "Fundamentals of Digital Logic with VHDL Design". Mc Graw Hill, pp. 1-6 y 899-916, second edition 2005.

[62] Luis Jacobo Álvarez Ruiz de Ojeda. "Historia de los Circuitos Digitales Configurables". Universidad de Vigo.
http://www.dte.uvigo.es/logica_programable/documentos/curso_disenho_digital_con_CDCs/Documento_historia_PLDs.pdf

[63] Miguel Ángel Freire Rubio. "Diseño Síncrono de Circuitos Digitales". Ingeniería Técnica de Telecomunicación, EUITT, Universidad Politécnica de Madrid, septiembre 2008.
http://www.euitt.upm.es/uploaded/464/DS_Sep_2008.pdf

[64] "Electrónica Digital, Tema 3, Diseño Síncrono". Departamento de sistemas electrónicos y de control, EUITT, Universidad Politécnica de Madrid, Curso 2010-2011.
http://www.euitt.upm.es/uploaded/464/teoria/tema3/Tema3.pdf

[65] Delmar Thomson. "Digital Design with Cpld Applications and VHDL". Thomson Delmar Learning, pp 275-298, 1ª edition June 28, 2000.

[66] Enoch O. Hwang. "Digital Logic and Microproccesor Design with VHDL". Brooks Cole, pp. 175-182, 2005.

[67] Documentación disponible en la ayuda de Xilinx Ise:
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/isehelp_start.htm

http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_c_configuration_overview.htm

http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_c_using_xst_for_synthesis.htm

[68] Documento Xilinx: "Synthesis and Simulation Design Guide". UG626 (v 13.3) October 19, 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sim.pdf

[69] Documento Xilinx: "ISE In-Depth Tutorial". UG695 (v13.3) October 19, 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_tutorial_ug695.pdf

[70] Página Web Altera: http://www.altera.com/

[71] Página Web IP Viterbi compiler de Altera: Enlace

[72] Altera: "Viterbi compiler User Guide". UG-Viterbi-11.0. Mayo 2011.
http://www.altera.com/literature/ug/ug_viterbi-compiler.pdf

[73] Página Web Lattice: http://www.latticesemi.com/

[74] Página Web IP block Viterbi decoder Lattice:
http://www.latticesemi.com/products/intellectualproperty/ipcores/blockviterbidecoder/index.cfm

[75] Lattice: "Block Viterbi Decoder User's Guide". IPUG32_02.7, Junio 2010.
http://www.latticesemi.com/lit/docs/ip/ipug32.pdf

[76] Página Web Actel: http://www.actel.com/

[77] Actel: "Convolutional Encoder and Viterbi Decoder Core Datasheet. For Actel FPGAs. http://www.actel.com/ipdocs/ViterbiDec_DS.pdf

[78] Página Web Colegio oficial ingenieros de telecomunicación: http://www.coit.es/ (Último acceso en abril 2012).

[79] "La práctica del ejercicio profesional por los ingenieros de telecomunicación". Normativa COIT, página 40, publicado en 2011. Referencia documento: 20110701_LaPracticaEjercicioProfesional2011_DTCOIT_v2

_http://www.coit.es/index.php?op=informaciong

http://www.coit.es/index.php?op=restringido&ref=%2Fdescargar.php%3Fidfichero%3D5481

http://acit.es/files/ACIT/documentos/la_practica_EP-2005.pdf

[80] Ley de colegios profesionales. Ley 25/2009 de 22 de diciembre, de modificación de diversas leyes para su adaptación a la Ley sobre el libre acceso a las actividades de servicios y su ejercicio. Publicada en BOE número 308 de 23/12/2009, páginas 108507 a 108578 (72 páginas). Referencia BOE-A-2009-20725.

__http://www.boe.es/aeboe/consultas/bases_datos/doc.php?id=BOE-A-2009-20725

CAPÍTULO 2

2. CARACTERÍSTICAS CODIFICADOR CONVOLUCIONAL, DECODIFICADOR VITERBI.

2.1 Objetivos.

En este tema aportamos los conocimientos teóricos que se necesitan para realizar el proyecto. Además hemos seleccionado una extensa bibliografía, *apartado 2.8*, donde se describen rigurosamente todos los aspectos relacionados con la codificación convolucional y la decodificación Viterbi. Nuestro objetivo en este tema es hacer un resumen del contenido de las fuentes bibliográficas. Para no hacer demasiado extenso este documento, sólo describiremos más detalladamente los aspectos relacionados con el proyecto. En los temas menos relacionados con el proyecto realizamos una descripción más breve, pero citamos la bibliografía donde se puede ampliar información.

Hay un aspecto en concreto que no desarrollamos: el cálculo teórico de la probabilidad de error de un decodificador Viterbi. El motivo se debe a que es un campo muy extenso y que no puede resumirse. De manera que para desarrollarlo la única opción sería copiar la información presente en la bibliografía. Entonces consideramos que no tiene sentido añadir un contenido tan extenso a este documento, cuando está perfectamente detallado en la bibliografía. Además hemos facilitado el trabajo al lector si quiere documentarse sobre este aspecto. Porque está desarrollado en múltiples referencias bibliográficas del *apartado 2.8*, y nosotros hemos seleccionado las que nos parecen más adecuadas en *2.8.1E*.

En cada uno de los apartados de este tema 2, haremos referencia a los artículos de la bibliografía, que a nuestro juicio, explican mejor ese apartado en concreto. Pero se trata de una clasificación flexible, que debe entenderse como una ayuda al lector. Porque la mayoría de las referencias bibliográficas son genéricas sobre la codificación convolucional y la decodificación Viterbi. De manera que el lector no necesita ceñirse obligatoriamente a la referencia que se le indique, sino que puede consultar prácticamente cualquiera del *apartado 2.8*. Es una forma más rápida de acceder a la información más clara y precisa. Porque, si no hubiésemos hecho esta clasificación de las referencias más adecuadas para un apartado, sería el propio lector quién tendría que buscar la información. Y como la bibliografía es muy extensa, la búsqueda podría ser lenta y nada sencilla.

2.2 Reseña histórica, ¿quién es Viterbi?

Andrew James Viterbi es ingeniero electrónico y empresario. Nació en Bérgamo, Italia, el 9 de marzo de 1935, de padres judíos y en 1939 emigró con ellos a los Estados Unidos en condición de refugiado. Su nombre original era Andrea, pero sus padres decidieron cambiarlo cuando obtuvo la nacionalidad estadounidense. Porque "Andrea" es considerado un nombre femenino en la mayoría de países de habla inglesa.

Viterbi estudió en la Boston Latin School, y en 1952 entró en el MIT, Massachusetts Institute of Technology, como estudiante de ingeniería electrónica. Allí coincidió, entre otros, con Claude Shannon, Norbert Wiener, Robert Fano y Bruno Rossi. Recibió sus títulos de grado: (S.B., "Bachelor of Science" y S.M.), en 1957 por el MIT. Después recibió su título de Doctor en comunicaciones digitales por la Universidad del Sur de California.

Viterbi fue profesor de ingeniería electrónica en la UCLA, Universidad de California en los Ángeles, y la UCSD, Universidad de California en San Diego. Es el inventor del algoritmo de Viterbi, en abril de 1967, su especificación original está en [41]. Se trata del algoritmo de máxima verosimilitud para la decodificación de datos codificados convolucionalmente.

Viterbi fue cofundador de Linkabit Corporation, junto con Irwin Jacobs en 1968, una pequeña contrata militar. También fue fundador de Qualcomm Inc. en 1985, así como presidente de la sociedad de capital riesgo: The Viterbi Group. El año 2000, Viterbi estaba situado el 386º en la lista Forbes 400 de los americanos más ricos, con un capital estimado de 640 millones de dólares.

En 2002, Viterbi dedicó el Centro de Computación Andrew Viterbi'52 a su escuela, Boston Latin School. El 2 de marzo de 2004, la Escuela de Ingeniería de la Universidad del Sur de California, fue renombrada a Escuela de Ingeniería Viterbi en su honor, después de haber realizado una donación de 52 millones de dólares.
En el año 2007, Viterbi recibió la Medalla Nacional USA de las Ciencias.

2.3 Alternativas para controlar los errores en un sistema de telecomunicación.

Existen varias alternativas:

1. FEC, Forward Error Correction, corrección de errores en el destino, a posteriori, o hacia adelante. Es un sistema que permite tanto la corrección como la detección de errores en el receptor, sin retransmitir la información original. No siempre se eliminarán por completo los errores en el receptor, no se conseguirá $BER_{Rx}=0$. El objetivo es mejorar la BER, obteniendo $BER_{RX} < BER_{TX}$.

2. ARQ, Automatic Repeat Request, petición automática de retransmisión. En este sistema se pueden detectar errores en el receptor, pero no corregirlos. De manera que cuando el receptor detecte un error en un mensaje, enviará una solicitud de retransmisión del mensaje al transmisor. Y al recibir esta petición, el transmisor reenviará el mensaje solicitado. Se emplea en redes datos. No se puede emplear en sistemas de tiempo real. El funcionamiento básico consiste en que cuando el receptor recibe un mensaje correcto, contesta al transmisor con un mensaje ACK positivo. ACK es Acknowledgement, acuse de recibo. Si por el contrario el mensaje recibido es erróneo, el receptor contesta enviando un acuse de recibo negativo NAK, No Acknowledgement

Un sistema ARQ es capaz de detectar más errores de los que un sistema FEC es capaz de corregir. Esto significa que en ocasiones FEC no puede corregir todos los errores en el receptor. En canales muy ruidosos, si se emplea FEC, no conseguiremos eliminar por completo los errores en el receptor, no conseguiremos $BER_{RX}=0$. El objetivo en un sistema FEC muy ruidoso es mejorar la BER, obteniendo $BER_{RX} < BER_{TX}$.

Con el sistema ARQ estaríamos más cerca de conseguir $BER_{RX}=0$ en el receptor. Sin embargo, como el canal es muy ruidoso, habría que hacer múltiples reenvíos de los mensajes. Esto disminuye mucho el ancho de banda del canal y se trata por tanto de una técnica poco eficiente. Por este motivo ARQ no se utiliza en solitario en canales muy ruidosos.

Por tanto, en los sistemas donde los errores no son admisibles, ejemplo redes de datos, se emplea ARQ. En estos sistemas ARQ garantiza casi al 100 % que el paquete recibido en el receptor es correcto. Mientras que con FEC la garantía de que el paquete es correcto sería menor. Por el contrario, en los sistemas donde los errores son admisibles, y lo que se necesita es mejorar la BER en el receptor frente a la del transmisor, se emplea FEC. Por ejemplo en sistemas en tiempo real, donde es preferible que llegue un mensaje con errores a tener que retransmitir ese mensaje. Un claro ejemplo de esta aplicación sería telefonía.

Las dos ventajas fundamentales de ARQ frente a FEC son:

1. El equipo de detección de errores ARQ es mucho más simple que el FEC.

2. Con ARQ se requiere menos redundancia en los códigos.

Los sistemas FEC se utilizan en sistemas en los que los mensajes erróneos no se pueden retransmitir:

1. Sistemas simplex TX→RX, sólo hay canal en el sentido transmisor→receptor y no hay canal de vuelta de receptor a transmisor. Esto imposibilita usar un esquema ARQ, puesto que no hay canal para enviar los mensajes ACK/NAK.

2. Sistemas en tiempo real, ejemplo telefonía, vídeo..., en los que no es posible la retransmisión porque no se admiten retardos en los mensajes.

3. Canales muy ruidosos en los que el sistema ARQ implicaría un excesivo número de retransmisiones.

También puede emplearse una técnica conjunta de FEC y ARQ, se denomina concatenación de códigos. En sistemas muy ruidosos, la técnica FEC disminuye los errores en el receptor, pero no consigue eliminarlos por completo. Por eso a continuación del FEC se utiliza un ARQ, que comprueba si el FEC ha conseguido eliminar por completo los errores en los mensajes. Si por el contrario ARQ descubre que el FEC no ha podido corregir uno o más errores, el mensaje se retransmite.

Los sistemas FEC se basan en añadir redundancia a la información original. Gracias a la cual se consiguen detectar y corregir los errores causados por el ruido en la transmisión-recepción. Hay varios tipos de codificadores-decodificadores:

- Códigos de bloque. Los más habituales son: Hamming, Cíclicos binarios, Golay, BCH, Reed Solomon o RS... La redundancia consiste en que al codificador llegan bloques compuestos por k bits de datos. Y a la salida del codificador se obtienen bloques de n bits, que es lo que se transmite. n es mayor que k, por tanto el codificador de bloque añade n-k bits redundantes. Si R_b es la tasa de bits de información a la entrada del codificador, la tasa de bits en su salida es: $R_c=nR_b/k$.

- Convolucional. La redundancia consiste en que por cada k bits de datos que entran al codificador se obtienen n en su salida. De manera que por cada k bits de datos se transmiten n bits. Siempre se cumple que n es mayor que k, por lo que el codificador convolucional añade redundancia. El término que determina la redundancia es la tasa o code rate: $R=k/n<1$. El algoritmo de máxima verosimilitud para decodificar un código convolucional es el decodificador Viterbi. Pero no es el único, pueden emplearse otros decodificadores sub-óptimos, consultar páginas 510-522 de [1], 422-430 de [2] y 62-64 de [12].

- Turbo códigos: Consisten en concatenar dos, o más, codificadores relativamente sencillos separados por un *interleaver*, (entrelazador). Lo más habitual es que los codificadores sean convolucionales sistemáticos recursivos, RSC (Recursive Systematic Convolutional). Pero también es posible utilizar otros tipos de codificadores. Con un turbo código, se consigue la misma corrección de errores que obtendríamos empleando un único codificador convolucional, de memoria igual a la profundidad del entrelazador. Pero la complejidad del proceso de codificación y decodificación del turbo código, es mucho menor que la complejidad necesaria para codificar y decodificar el codificador convolucional equivalente.

Los turbo códigos fueron inventados por Claude Berrou, Alain Glavieux y Punya Thitimasjshima. Y fueron presentados en 1993 en la conferencia internacional de la IEEE en Ginebra Suiza.

No entramos en más detalle describiendo los sistemas FEC ni ARQ, para ampliar información sobre estos sistemas están las referencias:

- Páginas 3-10 de [16].
- Páginas 41,42; 65-76 de [8].
- [37], [38], [39], [40].
- Cualquiera del *apartado 2.8.1*.

2.3.1 Diagrama de bloques de un sistema de comunicaciones FEC.

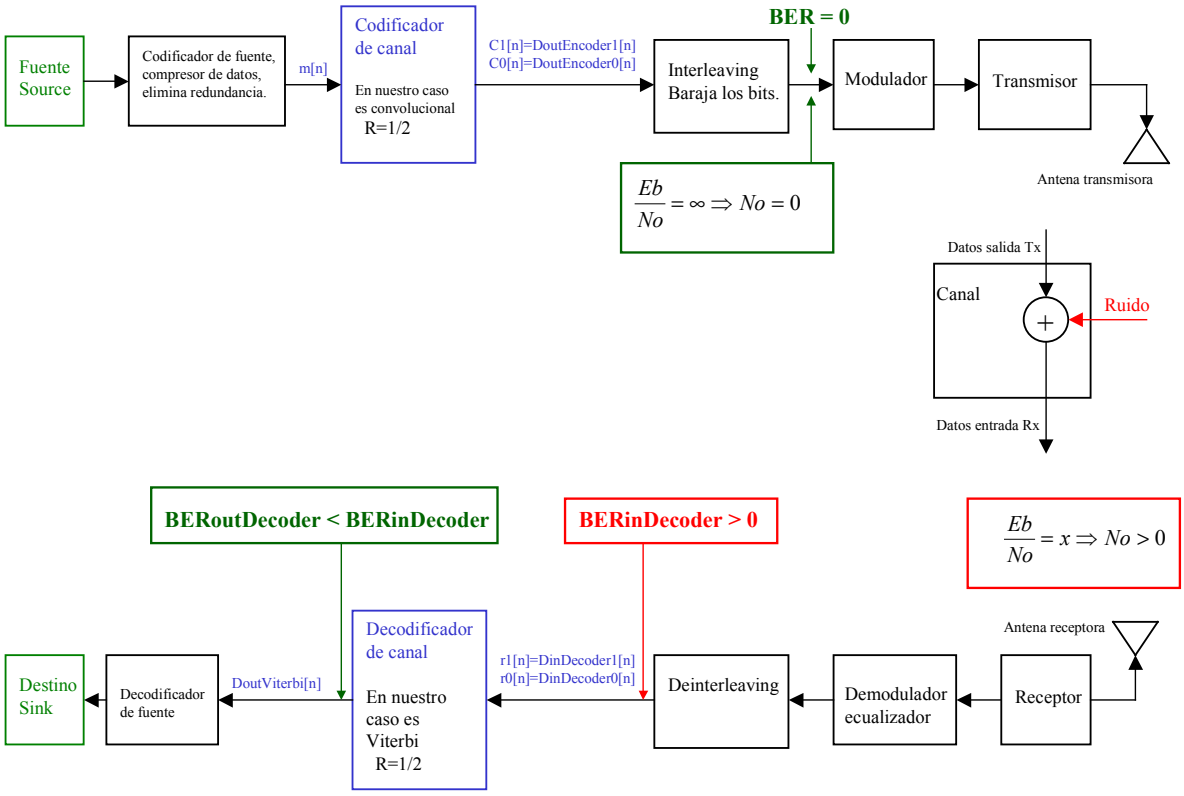


Figura 2.1 Diagrama de bloques sistema FEC.

El sistema de la figura anterior está particularizado para un codificador convolucional y un decodificador Viterbi, que son los que empleamos es este proyecto. Pero el esquema es válido para cualquier otro codificador-decodificador.

A continuación realizamos una breve descripción de los distintos módulos. No nos extendemos porque estos conceptos los trataremos en el *apartado 6.3*. Y además están ampliamente desarrollados en las referencias, del *tema 6*, *apartado 6.9*: [4, 5A, 6A, 7A, 8, 9, 10, 11A].

Fuente: Consiste en la información a transmitir, por ejemplo: un archivo digital, una secuencia de vídeo, una conversación telefónica... Se representa en forma digital, por lo que si la información original es analógica, debe convertirse en digital mediante conversores analógicos-digitales. La fuente se representa como unas cadenas de bits aleatorios con una determinada distribución de probabilidad.

Codificador de fuente: Comprime la información fuente, eliminando las redundancias para que pueda representarse utilizando menos bits. La codificación fuente puede ser con pérdidas y sin pérdidas:

1. La codificación sin pérdidas es reversible, de manera que si a la secuencia codificada le aplicamos un decodificador, se obtiene la secuencia original.
   Si la función codificadora es F y la decodificadora es F^{-1}, se cumple:
   $y[n] = F(x[n]) \quad \Rightarrow F^{-1}(y[n) = F^{-1}(F(x[n])) = x[n]$

2. La codificación con pérdidas no es reversible. De manera que si decodificamos la secuencia codificada, no obtendremos una igual a la original. Esto se emplea por ejemplo en vídeo e imagen, donde la información comprimida con pérdidas tiene menos calidad que la original.
   $y[n] = F(x[n]) \quad \Rightarrow F^{-1}(y[n) = F^{-1}(F(x[n])) = z[n] \neq x[n]$

Codificador de canal: Codifica la información a transmitir añadiendo redundancia. Gracias a esta redundancia, el decodificador situado en el receptor podrá detectar y corregir los errores producidos por el ruido presente en el sistema. En el caso particular de nuestro proyecto, utilizamos un codificador convolucional con tasa 1/2.
Este codificador convolucional puede sustituirse por uno de bloque, por una concatenación de codificadores convolucionales, o una concatenación de codificadores convolucionales y de bloque.

Interleaving, o intercalador, o entrelazador: Se encarga de mezclar, o barajar, los bits a transmitir, para eliminar o al menos minimizar, los errores de ráfaga. Los errores de ráfaga se deben al fading, (desvanecimiento selectivo en frecuencia). Al barajar los bits, los errores de ráfaga en la trama se convierten en errores uniformemente distribuidos en la trama. Así se consigue mejorar la probabilidad de error en recepción. Porque el codificador-decodificador de canal puede corregir los errores uniformemente distribuidos, sin embargo es mucho menos efectivo corrigiendo errores de ráfaga. En el *apartado 6.4.1* hemos desarrollado un ejemplo de entrelazador.

Modulador: Convierte las secuencias de bits, o símbolos, provenientes del entrelazador, en señales apropiadas para ser transmitidas por el canal.

Transmisor: Se encarga de amplificar la señal y adaptarla para ser transmitida por el canal. El dispositivo real añade ruido al sistema.

Canal: Medio físico sobre el que las señales recorren el camino entre el transmisor y el receptor. Las señales sufren modificaciones al atravesarlo. Por ejemplo: el canal añade ruido, atenúa la potencia de las señales, añade retardos, desfases, offsets, interferencias, reflexiones...
Ejemplos de canales físicos son: canales radio, canales de fibra óptica, líneas de cobre, microondas, canales de alta frecuencia...

Receptor: Recibe las señales presentes en el canal, las amplifica y las adapta para que sean la entrada del demodulador. El dispositivo real añade ruido al sistema.

Demodulador/ecualizador: Convierte las señales en una secuencia de bits, o símbolos. Realiza la función inversa al modulador.

***Deinterleaving* o desentrelazador**. Realiza la función inversa al *interleaving*. De manera que si la cadena de salida del entrelazador, fuese la entrada al desentrelazador, se obtendría una cadena exactamente igual a la entrada al entrelazador. Por tanto, el entrelazado es reversible sin pérdidas.

$$y[n] = Entrelazado(x[n]) = F(x[n])$$
$$\Rightarrow Desentrelazado(y[n]) = F^{-1}(y[n) = F^{-1}(F(x[n])) = x[n]$$

Decodificador de canal: Realiza la función inversa al codificador de canal. O a los codificadores, en caso de que se haya empleado una concatenación de codificadores de canal. En el caso particular de nuestro proyecto empleamos un decodificador Viterbi.

Decodificador de fuente: Realiza la función inversa al codificador de fuente.

Destino, sink: La información resultante de todo el proceso de telecomunicación.

En un sistema ideal, la fuente será exactamente igual al destino. Sin embargo en los sistemas reales, el transmisor, el canal y el receptor, perturban la señal original, porque no son dispositivos ideales. Las perturbaciones se deben: al ruido, atenuación de la potencia de las señales, retardos, desfases, offsets, interferencias, reflexiones... La consecuencia de esto es que se producen errores en la transmisión de las señales. De manera que en la *figura 2.1*, los bits, o símbolos, recibidos no son iguales a los transmitidos.

Para realizar nuestro proyecto, necesitamos implementar un simulador FEC, ver *tema 6*. En este simulador no es necesario implementar los bloques codificador de fuente, entrelazador y modulador, porque no aportan nada útil al simulador. Esto se debe a que su función se anula con la de sus bloques inversos: decodificador de fuente, desentrelazador y demodulador. Por lo que no modifican nada la secuencia de bits transmitida.

En el simulador no es necesario implementar el transmisor ni el receptor, porque el receptor realiza la función inversa al transmisor. Entonces es irrelevante para el proyecto el proceso de amplificación y adaptación de las señales. Sin embargo es fundamental modelar las perturbaciones que producen el transmisor, el canal y el receptor en las señales transmitidas. Estas perturbaciones ocasionan errores en el sistema de comunicaciones, de manera que los bits, o símbolos transmitidos, no son iguales a los recibidos. Este efecto lo implementamos en el simulador con una única fuente de ruido, que es la suma de todas las perturbaciones que producen los 3 equipos reales: transmisor, canal y receptor. Se trata de una técnica común al implementar o simular un sistema FEC, que consiste en reducir todas las fuentes de ruido a una sola que englobe los efectos de los tres módulos.

El funcionamiento del sistema de la *figura 2.1*, simplificándolo según las necesidades específicas de nuestro proyecto consiste en:

1. En la entrada al codificador tenemos la secuencia original, m[n].

2. La secuencia original se codifica, obteniéndose c[n] = ($DoutEncoder_1$[n], $DoutEncoder_0$[n]). Tanto a la entrada como a la salida del codificador la BER es cero, porque aún no se han producido errores.

3. C[n] pasa por el entrelazador, que baraja los bits a transmitir, para hacerlos más robustos frente a los errores de ráfaga. En el *apartado 7.8* simularemos un sistema FEC sin entrelazador-desentrelazador. Y mediante ese ejemplo, demostraremos de manera práctica porque es imprescindible el entrelazado en los sistemas FEC.

4. Las señales pasan por el transmisor, canal y receptor. Estos 3 módulos perturban la señal de manera que se producen algunos errores en los bits o símbolos transmitidos.

5. El desentrelazador realiza la función inversa al entrelazador.

6. Debido a los errores producidos en los módulos transmisor, canal y receptor, la secuencia de entrada al decodificador, es diferente a la de salida del codificador. r[n]≠ c[n]. BERinDecoder > 0.

7. El decodificador consigue corregir todos, o al menos una cantidad importante, de los errores producidos por el transmisor, el canal, y el receptor. De manera que el decodificador consigue mejorar la BER del sistema, obteniéndose BERoutDecoder = 0, o al menos BERoutDecoder < BERinDecoder.

2.4 Codificación convolucional.

2.4.1 Introducción.

Los códigos convolucionales fueron inventados en 1954 por Peter Elias. Constituyen una familia dentro de los códigos correctores de errores. Son muy utilizados en los sistemas FEC, porque presentan estas ventajas:

- Simplicidad en el decodificador.

- Un buen rendimiento corrector de errores, se consiguen ganancias de codificación altas. Sobre todo en canales modelados con ruido AWGN, Additive White Gaussian Noise.

En la referencia [42] está la especificación original de Peter Elias. Además disponemos de una extensa bibliografía sobre codificación convolucional, porque este campo constituye la base de nuestro proyecto. Por eso prácticamente todas las referencias del *apartado 2.8* describen la codificación convolucional. Para facilitar el acceso a la información hemos seleccionado las de los *apartados 2.8.1A* y *2.8.2* como las más representativas.

Los códigos convolucionales pueden utilizarse en varias estructuras de codificación diferentes:

1. Un codificador convolucional en solitario.
2. Un codificador de bloque y uno convolucional concatenados.
3. Dos, o más, codificadores convolucionales concatenados y separados por un entrelazador. Esto constituye un turbo código.

Un codificador convolucional trabaja sobre la estructura de un campo finito o Galois Field, GF(q). GF(q) constituye un campo finito en el que los números pueden tener q valores enteros, de 0 a q-1. El más habitual es el campo binario, GF(2), en el cual sólo existen dos valores: 0 y 1. En las páginas 193 a 207 de [1] se explican los campos GF(q).

En toda la documentación de nuestro proyecto, nos referiremos siempre a un campo binario GF(2).

En GF(2) la operación de suma se define de la siguiente manera:

- 0+0=0
- 0+1=1
- 1+0=1
- 1+1=0

Por tanto la operación de suma consiste en una operación XOR, OR exclusiva.

La estructura de un codificador convolucional puede definirse como un conjunto de filtros digitales, (sistemas lineales e invariantes en el tiempo). De este modo la operación codificadora es equivalente a una operación de filtrado o convolución. Las secuencias de código de salida son el resultado de sumar en el campo GF(2) las salidas de los filtros.

Los códigos convolucionales son ampliamente usados en la práctica, con diferentes implementaciones de hardware disponibles tanto para el codificador como para el decodificador.

Los códigos convolucionales tienen un excelente rendimiento cuando los comparamos con los códigos de bloque de complejidad codificador/decodificador equivalente. Además, fueron los primeros códigos en los que se implementaron algoritmos efectivos de decisión blanda, (*soft decision*). Por tanto estas ventajas justifican el empleo de codificación convolucional en multitud de aplicaciones prácticas.

Los códigos de bloque toman bloques discretos de k símbolos y a partir de ellos producen bloques de n símbolos, que dependen sólo de los k símbolos de entrada. En cambio en los convolucionales, la codificación no está dividida en bloques, sino que es continua, stream codes. Trabajan con un flujo continuo de símbolos que no están divididos en bloques finitos de mensajes. Aunque es importante indicar que también funcionan si la secuencia de símbolos de entrada no es continua, sino que está dividida en tramas.

En un código de bloque, la secuencia de bits de entrada debe estar obligatoriamente dividida en bloques de k bits. El codificador añade redundancia y obtiene bloques de n bits, donde n es mayor que k. Cada bloque de n bits obtenido en la salida depende únicamente del bloque de k bits del que procede, por lo tanto no tiene memoria. La relación k/n se denomina tasa o ratio del código, y es una medida de la redundancia que añade el codificador.

En un código convolucional, en cada instante de tiempo t_x hay un símbolo de k bits presente en la entrada del codificador, y se obtiene otro de n bits en la salida. La secuencia de símbolos de entrada es continua, por lo que en cada instante de tiempo: t_{x+1}, t_{x+2}, t_{x+3}... llegará un nuevo símbolo de k bits a la entrada y se obtendrá otro de n bits en la salida. Siempre se cumple $n > k$, por tanto la tasa o code rate del codificador es $R=k/n <1$. Este término indica la redundancia que añade el codificador.

El codificador convolucional tiene memoria, de manera que en cada instante t_x se obtiene un símbolo de n bits en la salida. Y ese símbolo depende de los k bits presentes en la entrada en t_x, y también de los k bits presentes en la entrada en t_{x-1}, t_{x-2} ... t_{x-M}. Donde M es el orden de memoria del codificador.

2.4.2 Estructura básica, parámetros que definen al codificador.

Los códigos convolucionales son códigos lineales que consisten en una serie de registros de desplazamiento (*shift registers*). Los bits de entrada al codificador entran en serie a los registros de desplazamiento. En cada instante de tiempo entran k bits al codificador y se obtienen n bits en la salida. Las salidas consisten en una combinación lineal en el campo GF(2) de los bits contenidos en los registros de desplazamiento y de los bits de entrada al codificador.

Un codificador convolucional está definido por los siguientes términos:

1. **k : Input frame**, es el número de bits que entran al codificador en cada instante de tiempo.

2. **n: Output frame**, es el número de bits que se obtienen en la salida del codificador en cada instante del tiempo.

3. **Tasa o code rate: R=k/n**.

4. **Función de transferencia**. Consiste en una matriz k*n. k filas * n columnas. Siempre se cumple n>k, porque el codificador añade redundancia.

$$G(x)=\begin{pmatrix} g_{11}(x) & g_{12}(x).....g_{1j}(x).....g_{1n}(x) \\ g_{21}(x) & g_{22}(x).....g_{2j}(x).....g_{2n}(x) \\ ..... \\ g_{i1}(x) & g_{i2}(x).....g_{ij}(x).....g_{in}(x) \\ ..... \\ g_{k1}(x) & g_{k2}(x).....g_{kj}(x).....g_{kn}(x) \end{pmatrix}$$

$g_{ij}(x)$ puede ser una función polinómica o una función racional. Es la respuesta al impulso del codificador conectando la entrada i con la salida j-1.

Mediante la función de transferencia es inmediato encontrar la relación entre los bits de entrada y los de salida:

$c(x)=m(x)G(x)$
$c(x)$ consta de n bits, $c_0(x)....c_{n-1}(x)$
$m(x)$ consta de k bits, $m_1(x)....m_k(x)$

$c_0(x)=m_1(x)*g_{11}(x) + m_2(x)*g_{21}(x) + ……. + m_i(x)*g_{i1} +…….+ m_k(x)*g_{k1}(x)$
$c_1(x)=m_1(x)*g_{12}(x) + m_2(x)*g_{22}(x) + ……. + m_i(x)*g_{i2} +…….+ m_k(x)*g_{k2}(x)$
$c_j(x)=m_1(x)*g_{1j+1}(x) + m_2(x)*g_{2j+1}(x) + …. + m_i(x)*g_{ij+1} +……..+ m_k(x)*g_{kj+1}(x)$
$c_{n-1}(x)=m_1(x)*g_{1n}(x) + m_2(x)*g_{2n}(x) + ……+ m_i(x)*g_{in} +…….+ m_k(x)*g_{kn}(x)$

El símbolo * representa convolución discreta en el dominio del tiempo discreto [n] y multiplicación en el dominio polinómico (x). Todas las operaciones son módulo 2, campo GF(2).

5. **Memoria completa del codificador, puede denominarse m, ó v**, dependiendo de la bibliografía consultada. Consiste en la suma del número total de registros del codificador.

v_i es el número máximo de etapas de registros de desplazamiento que hay en el camino desde la entrada i hasta cualquiera de las salidas. Es la longitud del registro de desplazamiento más largo asociado a la entrada i. En esta longitud o camino no se incluyen los registros que se utilicen solamente para registrar la entrada o la salida al codificador. Sólo se cuentan los registros de desplazamiento.

v_i se obtiene mediante la función de transferencia. Consiste en el número de registros en una fila de la función de transferencia del codificador. Su valor es igual al del grado del polinomio de mayor grado en la fila i de la función de transferencia.

$$m = v = \sum_{i=1}^{k} v_i \qquad \text{Ecuación 1.}$$

$$v_i = \max_j \deg\left(g_{ij}(x)\right)$$

6. **Número de estados**: El codificador es una máquina de estados finita, en la que se cumple: número de estados = $2^{v}=2^{m}$.

Las definiciones 7 y 8 sólo se aplican a los codificadores no recursivos, *apartado 2.4.3*.

Todo codificador recursivo tiene un codificador no recursivo equivalente. Esto se demuestra con un teorema que indica: "Every rational encoder has an equivalent basic transfer function matrix". El teorema puede consultarse en las páginas 464-465 de [1].

La consecuencia del teorema es que cualquier código convolucional puede representarse mediante un codificador no recursivo. Por tanto las definiciones 7 y 8 son aplicables a cualquier codificador, aunque si este es recursivo, antes habrá que obtener el no recursivo equivalente.

Pero hay un detalle importante en el teorema. Siempre existirá un codificador no recursivo y no sistemático equivalente, pero puede que no exista un no recursivo sistemático equivalente. Esto es importante, porque los turbo códigos suelen emplear códigos sistemáticos. De manera que habrá veces en los que la única opción para conseguir el código sistemático sea utilizar un codificador recursivo.

Mediante la función de transferencia se definen los distintos términos que caracterizan un codificador convolucional. Es importante indicar que no hay unanimidad en los términos m, v y M. De manera que otras fuentes bibliográficas pueden utilizar una notación diferente para referirse a ellos. En este documento utilizamos la más común, pero al consultar otros documentos debe tenerse en cuenta que pueden emplear una notación diferente para estas definiciones.

7. **Memory order = M**, es la longitud del registro de desplazamiento más largo.

$$M = \max_i(v_i) = \max_{ij} \deg\left(g_{ij}(x)\right)$$

8. **Constraint length**. Este término puede crear confusión al consultar la bibliografía, porque no sólo no hay unanimidad en la notación, sino que además su definición no es la misma en toda la bibliografía. Existen tres maneras diferentes para referirse a este término:

   a. **Memory constraint length = ν ó m**. Es el número total de etapas de registros de desplazamiento en el codificador. Excluyendo los registros que se utilicen únicamente para registrar las entradas o salidas. Este término ya lo hemos definido anteriormente en la ecuación 1: $m = \nu = \sum_{i=1}^{k} \nu_i$

   b. **Input constraint length = K**. Es el número total de bits involucrados en la operación de codificación: $K = \nu + k = \sum_{i=1}^{k} (1 + \nu_i)$

   c. **Constraint length =K.** El número máximo de bits involucrados en la codificación de una única salida del codificador. Es igual a la longitud del registro de desplazamiento más largo, más uno. También es igual al mayor grado de entre los polinomios g_{ij} más uno. $K = 1 + \max_i(\nu_i) = 1 + \max_{ij}(g_{ij}(x))$

El mayor inconveniente consiste en que en la bibliografía no se suele distinguir entre input constraint length ni memory constraint length. Los distintos documentos suelen emplear únicamente el término constraint length, al que se refieren como K ó ν. Además pueden emplear cualquiera de las tres definiciones anteriores tanto para referirse a K como a ν. Por tanto al consultar un documento es necesario tener claro cuál es la notación y la definición que se utiliza para el constraint length.

En este proyecto siempre utilizamos la definición c: constraint length es la longitud del registro de desplazamiento más largo más uno: $K = 1 + \max_i(\nu_i) = 1 + \max_{ij}(g_{ij}(x))$

Utilizamos esta definición en todos los capítulos de este documento y en todo el código que hemos desarrollado.

Un código convolucional suele identificarse con sólo 3 parámetros. Pero no hay unanimidad en la notación, las tres maneras más habituales de identificación son las dos siguientes:

1. $C_{conv}(n, k, K)$
2. $C_{conv}(n, k, m \text{ ó } \nu)$
3. $C_{conv}(n/k, K)$

En este documento utilizaremos siempre la primera opción. Pero de nuevo al consultar la bibliografía, hay que tener en cuenta que se puede utilizar una identificación diferente.

2.4.3 Codificadores no recursivos (FIR) o recursivos (IIR).

Si todas las funciones $g_{ij}(x)$ son polinómicas, entonces el codificador es no recursivo. También se denomina *FIR, (Finite Impulse Response),* o *feedforward.*
En un codificador no recursivo, no hay ninguna realimentación en ninguna etapa de los registros de desplazamiento. Ejemplos 1, 2, 3 y 4 de *2.4.6*.

Conque haya un sólo $g_{ij}(x)$ que sea racional, entonces el codificador es recursivo. También se denomina *IIR, (Infinite Impulse Response),* o *feedback*.
En un codificador recursivo hay realimentación. Los bits de salida se realimentan a la entrada, o alguna etapa de algún registro de desplazamiento se realimenta a una etapa anterior del registro de desplazamiento. Ejemplos 5 y 6 de *2.4.6*.

2.4.4 Codificadores sistemáticos o no sistemáticos.

Un codificador sistemático es aquel en el que la entrada aparece explícitamente en la salida. Todos los bits de entrada deben aparecer explícitamente en la salida. Así que cada uno de los puertos de entrada al codificador debe estar conectado directamente con un puerto de salida. Un codificador es sistemático si se puede identificar la matriz identidad entre los elementos de G(x). Ver ejemplos 4, 5 y 6 de *2.4.6*.

Si no se cumplen las propiedades anteriores, el codificador es no sistemático. Si k es igual a uno, entonces el codificador es no sistemático cuando no hay conexión directa entre ese puerto de entrada y una salida. Si k es mayor que uno, entonces el codificador es no sistemático cuando hay al menos un puerto de entrada que no está conectado directamente con una salida. Ver ejemplos 1, 2 y 3 de *2.4.6*.

2.4.5 Notación de la función codificadora.

En los codificadores no recursivos es habitual representar los polinomios $g_{ij}(x)$ que conectan la entrada i con la salida j-1, como vectores que representan la respuesta al impulso del codificador. Suele utilizarse notación binaria u octal. También puede utilizarse la representación dependiente del tiempo discreto, [n].

En la siguiente tabla mostramos las distintas notaciones que empleamos en este documento. Utilizamos el ejemplo del codificador de la *figura 2.3* :

Tabla 2.1: Posibles representaciones de la relación entre la entrada y la salida.		
Función de transferencia polinómica, válida para no recursivos y recursivos.	$G(x)= [1+x^2+x^3+x^5+x^6 \quad 1+x+x^2+x^3+x^6]$ $c_0(x)=m(x)*(1+x^2+x^3+x^5+x^6)=m(x)*g_{11}(x)$ $c_1(x)=m(x)*(1+x+x^2+x^3+x^6)= m(x)*g_{12}(x)$	
g_{11} y g_{12} son los polinomios generadores del código. Pueden usarse en no recursivos y recursivos.	$g_{11}(x)= 1+x^2+x^3+x^5+x^6$ $g_{12}(x)= 1+x+x^2+x^3+x^6$	
Vectores representando la respuesta al impulso, g_{11} y g_{12} son las secuencias generadoras del código. Sólo para codificadores no recursivos.	Binario	Octal
	g_{11}= [1011011] g_{12} =[1111001]	g_{11}= 133 g_{12}= 171
Representación matemática exacta, dependiente del tiempo discreto. Válida para no recursivos y recursivos.	$c_0[n] = m[n]+m[n-2]+m[n-3]+m[n-5]+m[n-6]$ $c_1[n] = m[n]+m[n-1]+m[n-2]+m[n-3]+m[n-6]$	

2.4.6 Ejemplos de codificadores con diferentes características.

El objetivo de este apartado es mostrar varios codificadores con diferentes parámetros. Cubrimos todas las posibilidades, que son:

1. No sistemático, no recursivo, ejemplos 1, 2 y 3.
2. Sistemático no recursivo, ejemplo 4.
3. Sistemático recursivo, ejemplos 5 y 6.

La opción no sistemático recursivo no la incluimos porque no hemos encontrado ningún ejemplo en la bibliografía. Estos codificadores no suelen utilizarse en la práctica, porque como vimos en el *apartado 2.4.2*, un código no sistemático siempre puede representarse mediante un codificador no recursivo. De manera que el no sistemático recursivo puede emplearse, pero en la práctica se suele emplear el no sistemático no recursivo equivalente.

Mostramos los ejemplos más representativos, que cubren gran parte de variaciones en los parámetros característicos de un codificador convolucional. En la bibliografía de *2.8.1* hay más ejemplos que no incluimos porque no son necesarios para el desarrollo ni la explicación de este proyecto.

Ejemplo 1, $C_{conv}(2, 1, K=3)$, no sistemático no recursivo.

Aparece en todas las referencias de *2.8.1* y *2.8.2*. Es un codificador sencillo con 4 estados, y por ello se utiliza ampliamente como ejemplo para describir las características de los codificadores convolucionales. Su capacidad de corrección de errores es baja, pero es muy útil para explicar la teoría de la codificación-decodificación Viterbi.

Para nosotros este codificador es particularmente útil, porque tiene una estructura muy similar a la del que implementamos en el proyecto. Ambos son no sistemáticos, no recursivos, con R=1/2. Se diferencian en el número de estados, 64 frente a 4, y en el polinomio generador.

En el codificador de 64 estados, no tiene sentido representar gráficamente la malla trellis, ni el autómata con la transición entre estados, ni el proceso de su decodificación Viterbi... Porque debido a su complejidad, tanto su desarrollo como su comprensión serían muy difíciles.

Por este motivo, en muchas partes de este documento nos basamos en este codificador sencillo de 4 estados para explicar: la malla trellis, el autómata con la transición entre estados, la decodificación Viterbi, un sistema completo FEC... Una vez conocido el proceso con este codificador, es inmediato aplicar estos conocimientos en el sistema real con 64 estados, porque ambos procesos tienen una estructura equivalente. Por eso nos basamos en este ejemplo en los *apartados: 4.6.2, 5.4.1, 5.5.10...*

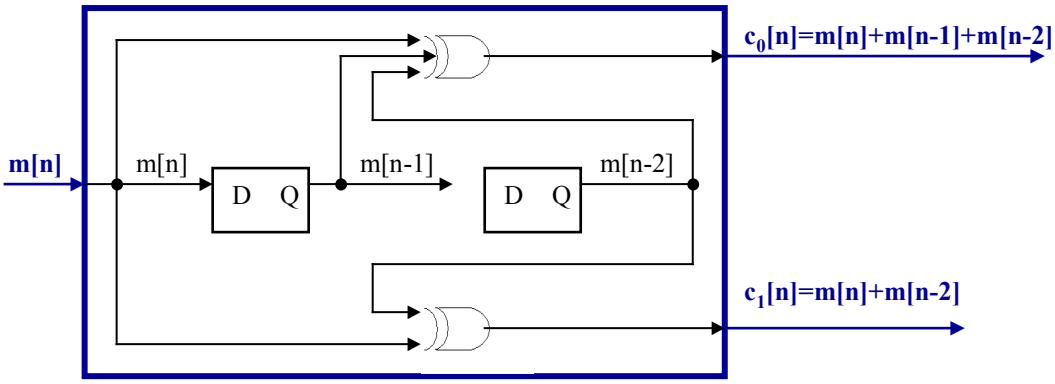


Figura 2.2: Codificador convolucional(2, 1,K=3), no sistemático, no recursivo.

Tabla 2.2: Parámetros codificador figura 2.2.			
Bits en la entrada = k	1		
Bits en la salida = n	2		
Tasa o rate R = k/n	½		
Función de transferencia polinómica.	$G(x) = [1+x+x^2 \quad 1+x^2]$ $c_0(x) = m(x)*g_{11}(x) = m(x)*(1+x+x^2)$ $c_1(x) = m(x)*g_{12}(x) = m(x)*(1+x^2)$		
Polinomios generadores de código	$g_{11}(x)= 1+x+x^2$; $g_{12}(x)= 1+x^2$		
Vectores representando la respuesta al impulso. Secuencias generadoras de código	Binario	Octal	
	$g_{11} = [111]$; $g_{12} = [101]$	$g_{11} = 7$; $g_{12} = 5$	
Representación exacta Mediante tiempo discreto	$c_0[n] = m[n]+m[n-1]+m[n-2]$ $c_1[n] = m[n]+m[n-2]$		
Memory order=M $M = \max_i(v_i)$ $v_i = \max_j \deg(g_{ij}(x))$	2		
Constraint length	$K = 1 + \max_{ij}(g_{ij}(x))$	Memory constraint length $m = v = \sum_{i=1}^{k} v_i$	Input constraint length K= ν + k
	3	2	3
Memoria completa $m = v = \sum_{i=1}^{k} v_i$	2		
Número de estados=$2^{v}=2^{m}$	4		

Ejemplo 2, $C_{conv}(2, 1, K=7)$, no sistemático no recursivo.

Se trata del codificador que implementamos en este proyecto y que desarrollamos en el *tema 3*. Se emplea en múltiples aplicaciones FEC. Es el estándar industrial para codificadores R = 1/2 y constraint length = 7. Es compatible con los siguientes estándares: Q1900, DVB, IEEE802.11a, IEEE802.16a, HiperAccess, HiperMAN, INTELSAT IESS-308/309.

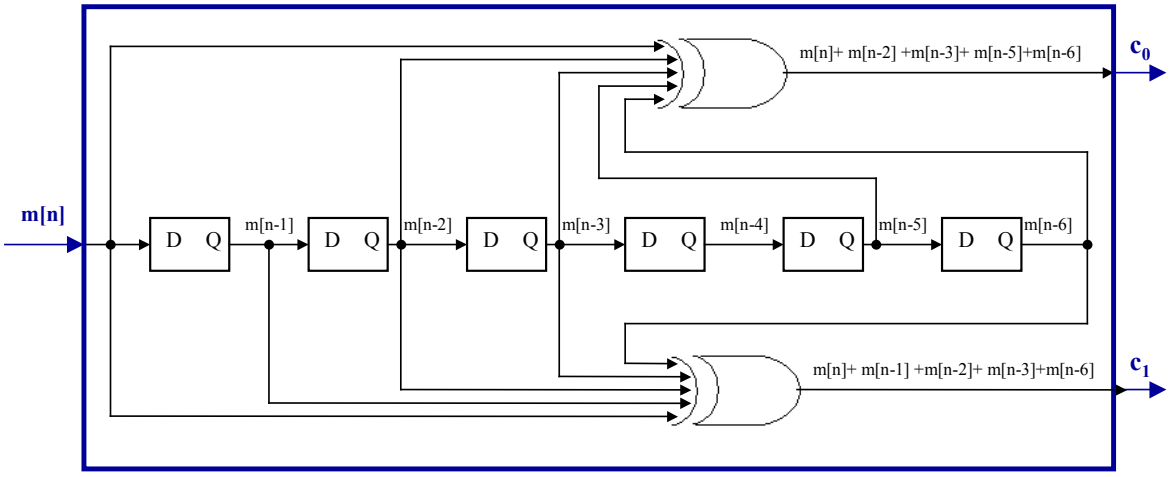


Figura 2.3: Codificador convolucional(2 , 1 ,K=7), no sistemático, no recursivo.

Tabla 2.3: Parámetros codificador figura 2.3.			
Bits en la entrada = k	1		
Bits en la salida = n	2		
Tasa o rate R = k/n	½		
Función de transferencia polinómica.	$G(x) = [1+x^2+x^3+x^5+x^6 \quad 1+x+x^2+x^3+x^6]$ $c_0(x) = m(x)*g_{11}(x) = m(x)*(1+x^2+x^3+x^5+x^6)$ $c_1(x) = m(x)* g_{12}(x) = m(x)*(1+x+x^2+x^3+x^6)$		
Generadores de código.	$g_{11}(x)= 1+x^2+x^3+x^5+x^6$; $g_{12}(x)= 1+x+x^2+x^3+x^6$		
Vectores representando la respuesta al impulso.	Binario	Octal	
	$G_{11} = [1011011]$; $g_{12} = [1111001]$	$g_{11} = 133$; $g_{12} = 171$	
Representación exacta Mediante tiempo discreto	$c_0[n] = m[n]+m[n-2]+m[n-3]+m[n-5]+m[n-6]$ $c_1[n] = m[n]+m[n-1]+m[n-2]+m[n-3]+m[n-6]$		
Memory order = M $M = \max_i(v_i)$ $v_i = \max_j \deg(g_{ij}(x))$	6		
Constraint length	$K = 1 + \max_{ij}(g_{ij}(x))$	Memory constraint length $m = v = \sum_{i=1}^{k} v_i$	Input constraint length K= ν + k
	7	6	7
Memoria completa $m = v = \sum_{i=1}^{k} v_i$	6		
Número de estados=$2^{v}=2^{m}$	64		

Ejemplo 3, C_{conv}(3, 2, K=3), no sistemático no recursivo.

Este codificador aparece en [43].

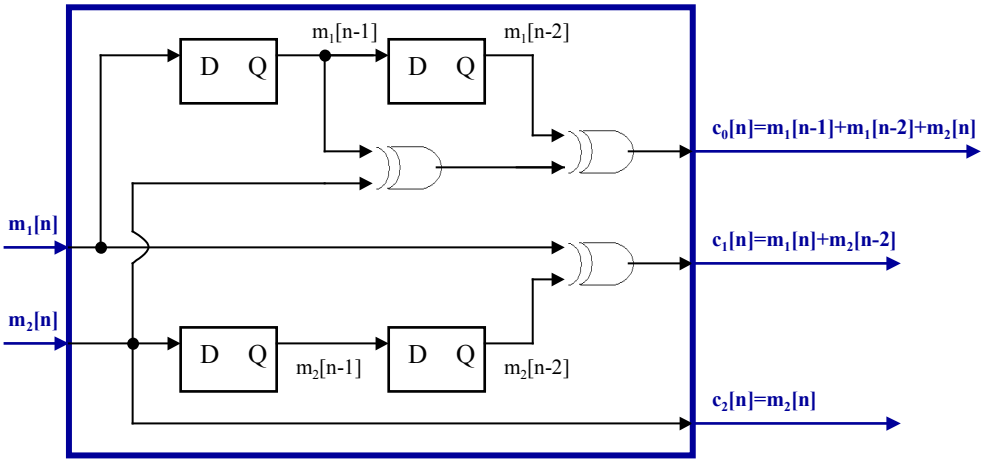


Figura 2.4: Codificador convolucional(3, 2, K=3), no sistemático, no recursivo.

Tabla 2.4: Parámetros codificador figura 2.4.			
Bits en la entrada = k	2		
Bits en la salida = n	3		
Tasa o rate R = k/n	2/3		
Función de transferencia polinómica.	$G(x)=\begin{pmatrix} x+x^2 & 1 & 0 \\ 1 & x^2 & 1 \end{pmatrix}$ $c_0(x) = m_1(x)*g_{11}(x)+m_2(x)*g_{21}(x) = m_1(x)*(x+x^2) + m_2(x)$ $c_1(x) = m_1(x)*g_{12}(x)+m_2(x)*g_{22}(x) = m_1(x) + m_2(x)*x^2$ $c_2(x) = m_1(x)*g_{13}(x)+m_2(x)*g_{23}(x) = m_2(x)$		
Generadores de código	$g_{11}(x) = x+x^2$; $g_{12}(x) = 1$; $g_{13}(x) = 0$; $g_{21}(x) = 1$; $g_{22}(x) = x^2$; $g_{23}(x) = 1$		
Secuencias generadoras de código	Binario		Octal
	$g_{11} = [011]$; $g_{12} = [100]$; $g_{13} = [000]$; $g_{21} = [100]$; $g_{22} = [001]$; $g_{23} = [100]$;		$g_{11} = 3$; $g_{12} = 4$; $g_{13} = 0$; $g_{21} = 4$; $g_{22} = 1$; $g_{23} = 4$;
Representación exacta mediante tiempo discreto	$c_0[n] = m_1[n-1]+m_1[n-2]+m_2[n]$ $c_1[n] = m_1[n]+m_2[n-2]$ $c_2[n] = m_2[n]$		
Memory order=M $M = \max_i(v_i)$ $v_i = \max_j \deg(g_{ij}(x))$	2		
Constraint length	$K = 1+\max_{ij}(g_{ij}(x))$	Memory constraint length $m = v = \sum_{i=1}^{k} v_i$	Input constraint length K= ν + k
	3	4 = ν = ν_1+ ν_2=2+2	6
Memoria completa = m= ν $m = v = \sum_{i=1}^{k} v_i$	4 = ν = ν_1+ ν_2; $\nu_1 = \nu_2 = 2$		
Número de estados=$2^{\nu}=2^m$	16		

Ejemplo 4, C_{conv}(3, 2, K=2), sistemático no recursivo.

Aparece en las páginas 129-134 de [13]

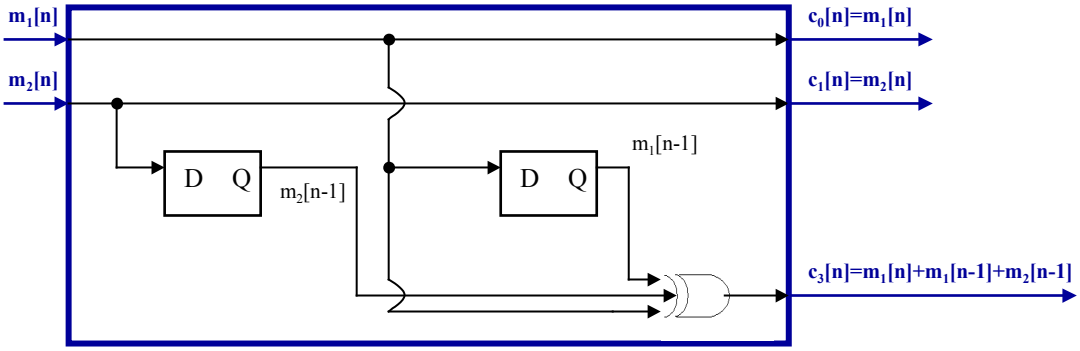


Figura 2.5: Codificador convolucional(3, 2, K=2), sistemático, no recursivo.

Tabla 2.5: Parámetros codificador figura 2.5.			
Bits en la entrada = k	2		
Bits en la salida = n	3		
Tasa o rate R = k/n	2/3		
Función de transferencia polinómica.	$G(x)=\begin{pmatrix}1 & 0 & 1+x\\ 0 & 1 & x\end{pmatrix}$ $c_0(x) = m_1(x)*g_{11}(x)+m_2(x)*g_{21}(x) = m_1(x)$ $c_1(x) = m_1(x)*g_{12}(x)+m_2(x)*g_{22}(x) = m_2(x)$ $c_2(x) = m_1(x)*g_{13}(x)+m_2(x)*g_{23}(x) = m_1(x)*(1+x)+m_2(x)*x$		
Polinomios generadores de código	$g_{11}(x) = 1$; $g_{12}(x) = 0$; $g_{13}(x) = 1+x$; $g_{21}(x) = 0$; $g_{22}(x) = 1$; $g_{23}(x) = x$		
Secuencias generadoras de código	Binario	Octal	
	$g_{11} = [10]$; $g_{12} = [00]$; $g_{13} = [11]$; $g_{21} = [00]$; $g_{22} = [10]$; $g_{23} = [01]$;	$g_{11} = 2$; $g_{12} = 0$; $g_{13} = 3$; $g_{21} = 0$; $g_{22} = 2$; $g_{23} = 1$;	
Representación exacta mediante tiempo discreto	$c_0[n] = m_1[n]$; $c_1[n] = m_2[n]$; $c_3[n] = m_1[n] + m_1[n-1]+m_2[n-1]$		
Memory order=M $M=\max_i(v_i)$ $v_i=\max_j \deg(g_{ij}(x))$	1		
Constraint length	$K=1+\max_{ij}(g_{ij}(x))$	Memory constraint length $m=v=\sum_{i=1}^{k} v_i$	Input constraint length K= ν + k
	2	2	4
Memoria completa $m=v=\sum_{i=1}^{k} v_i$	$2 = \nu = \nu_1+ \nu_2$; $\nu_1 = \nu_2 = 1$		
Número de estados=$2^{\nu}=2^m$	4		

Ejemplo 5, C_{conv}(2, 1, m=2), sistemático recursivo.

Este codificador aparece en las páginas 452-456 de [1], en el *anexo B1* detallamos todo el desarrollo matemático.

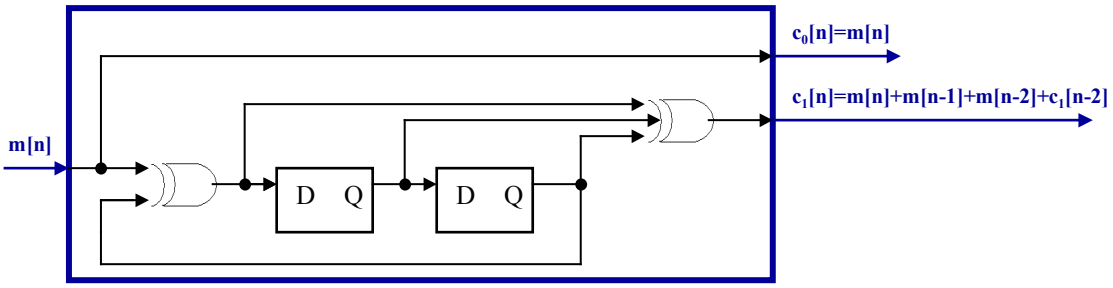


Figura 2.6: Codificador convolucional(2, 1, m=2), sistemático, recursivo.

Tabla 2.6: Parámetros codificador figura 2.6.	
Bits en la entrada = k	1
Bits en la salida = n	2
Tasa o rate R = k/n	½
Función de transferencia polinómica.	$G(x) = \left(1 \qquad \frac{1+x+x^2}{1+x^2}\right)$ $c_0(x) = m(x)*g_{11}(x) = m(x)$ $c_1(x) = m(x) * g_{12}(x) = m(x) * \frac{1+x+x^2}{1+x^2}$
Polinomios generadores de código	$g_{11}(x)= 1$; $g_{12}(x) = \frac{1+x+x^2}{1+x^2}$
Secuencias generadoras de código	No aplica por ser recursivo.
Representación exacta mediante tiempo discreto	$c_0[n] = m[n]$; $c_1[n] = m[n]+m[n-1]+m[n-2]+c_1[n-2]$
Memory order=M $M = \max_i(v_i)$ $v_i = \max_j \deg(g_{ij}(x))$	2
Constraint length	No aplica por ser recursivo.
Memoria completa $m = v = \sum_{i=1}^{k} v_i$	2 = ν
Número de estados=$2^{\nu}=2^m$	4

Ejemplo 6, C_{conv}(3, 2, m=3), sistemático recursivo.

Este codificador aparece en las páginas 452-456 de [1], en el *anexo B2* detallamos todo el desarrollo matemático.

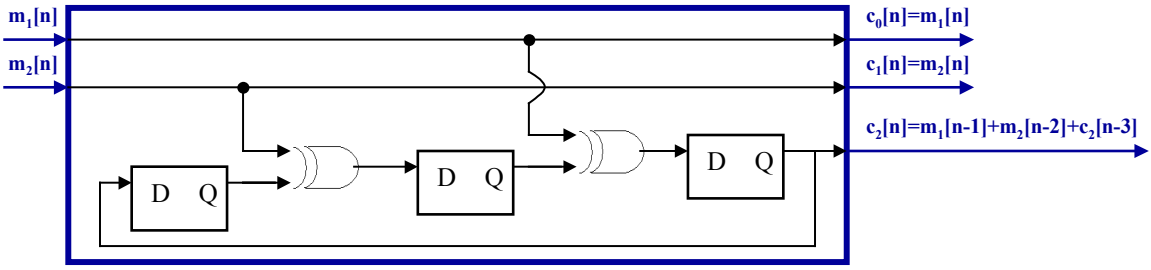


Figura 2.7: Codificador convolucional(3, 2, m=3), sistemático, recursivo.

Tabla 2.7: Parámetros codificador figura 2.7.	
Bits en la entrada = k	2
Bits en la salida = n	3
Tasa o rate R = k/n	2/3
Función de transferencia polinómica.	$G(x)=\begin{pmatrix}1 & 0 & \frac{x}{1+x^3}\\ 0 & 1 & \frac{x^2}{1+x^3}\end{pmatrix}$ $c_0(x) = m_1(x)*g_{11}(x)+m_2(x)*g_{21}(x) = m_1(x)$ $c_1(x) = m_1(x)*g_{12}(x)+m_2(x)*g_{22}(x) = m_2(x)$ $c_2(x)=m_1(x)*g_{13}(x)+m_2(x)*g_{23}(x)=m_1(x)*\frac{x}{1+x^3}+m_2(x)*\frac{x^2}{1+x^3}$
Polinomios generadores de código	$g_{11}(x)=1;\quad g_{12}=0;\quad g_{13}=\frac{x}{1+x^3}$; $g_{21}=0;\quad g_{22}=1;\quad g_{23}=\frac{x^2}{1+x^3}$
Secuencias generadoras de código	No aplica por ser recursivo.
Representación exacta mediante tiempo discreto	$c_0[n] = m_1[n];\quad c_1[n] = m_2[n];$ $c_2[n] = m_1[n-1]+m_2[n-2]+c_2[n-3]$
Memory order=M $M=\max_i(v_i)$ $v_i=\max_j \deg(g_{ij}(x))$	3 =
Constraint length	No aplica por ser recursivo.
Memoria completa $m=v=\sum_{i=1}^{k} v_i$	$3 = \nu_1+\nu_2;\quad \nu_1=1\ ;\ \nu_2=2$
Número de estados=$2^{\nu}=2^m$	8

2.4.7 Maquina de estados.

Un codificador convolucional es una máquina de estados. Como hemos visto anteriormente, si la memoria completa del codificador es de m ó ν registros, entonces tiene $2^m = 2^\nu$ estados.

Para implementar un codificador y su decodificador asociado, es fundamental conocer los diferentes estados y la transición entre ellos: paso del estado actual al siguiente. Para ello se pueden emplear tres representaciones diferentes:

1. Diagrama de estados. Se representan los diferentes estados y la relación entre ellos. Se indica cuál es el estado siguiente para cada uno de los estados, en función de las entradas y salidas al codificador. *Figura 2.8.* De cada nodo del diagrama parten 2^k ramas y llegan a él 2^k ramas.

2. Malla trellis, *figura 2.9*. Cada estado se representa con un nodo o vértice. Todos los estados del codificador se agrupan en una columna, que consiste en los estados del codificador en el instante de tiempo t_x. La columna siguiente, a su derecha, corresponde al siguiente estado, en t_{x+1}. Desde cada uno de los nodos o vértices en t_x, parten 2^k ramas, hacia los nodos o vértices que representen sus siguientes estados en t_{x+1}. Las ramas siempre van de t_x a t_{x+1}, no se puede dar otra combinación. El paso de t_x a t_{x+1} se denomina etapa. En cada una de las ramas se indica la entrada al codificador que hace pasar de un estado al siguiente, y la salida que se obtiene en t_x. Una malla trellis tiene un número de etapas igual al número de columnas de estados –1.

3. Tabla que contiene explícitamente la información estado actual-siguiente estado, consultar *tabla 2.8*.

En nuestro proyecto utilizamos el codificador del ejemplo 2. Sin embargo no representamos sus estados de ninguna de las tres maneras mencionadas. Porque al tener 64, emplearíamos mucho tiempo en representarlos. Otro inconveniente aún más grave consiste en que esta representación sería muy compleja, con lo que resultaría muy difícil trabajar con ella para comprender la metodología del codificador-decodificador.

Por este motivo trabajamos con el diagrama de estados y la malla trellis del ejemplo 1. Al tener 4 estados, su representación es sencilla y se puede trabajar fácilmente con ella, porque los gráficos, diagramas y tablas que se generan son simples. La mayor ventaja de este sistema consiste en que los conocimientos y metodología que aplicamos sobre este codificador-decodificador, se pueden aplicar directamente sobre el que utilizamos en el proyecto. Porque ambos tienen la misma estructura: no sistemático, no recursivo, R=1/2.

A continuación calculamos las transiciones entre estados y las salidas para el codificador del ejemplo 1.

Estado inicial 00, llega un '0'.

t_x corresponde a n=0, por lo que en la entrada del codificador tenemos m[n]=m[0]='0'.
Este estado en t_x, n=0, corresponde a: m[n-1]=m[-1]='0'; m[n-2]=m[-2] = '0'.

Las salidas correspondientes en t_x son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '0'+'0'+'0'='0'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '0'+'0'='0'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado. Su cálculo es inmediato, se trata de un registro de desplazamiento en el que los bits almacenados en los registros se desplazan una posición en cada unidad de tiempo. De manera que:
m[n-1]= m[0]='0'; m[n-2]=m[-1] = '0'; estado "00".

Estado inicial "00", llega un '1'.

Entrada del codificador: m[n]=m[0]='1'.
En t_x, n=0, el estado es "00": m[n-1]=m[-1]='0'; m[n-2]=m[-2] = '0'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '1'+'0'+'0'='1'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '1'+'0'='1'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='1'; m[n-2]=m[-1] = '0'; estado "10".

Estado inicial "01", llega un '0'.

Entrada del codificador: m[n]=m[0]='0'.
En t_x, n=0, el estado es "01": m[n-1]=m[-1]='0'; m[n-2]=m[-2] = '1'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '0'+'0'+'1'='1'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '0'+'1'='1'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='0'; m[n-2]=m[-1] = '0'; estado "00".

Estado inicial "01", llega un '1'.

Entrada del codificador: m[n]=m[0]='1'.
En t_x, n=0, el estado es "01": m[n-1]=m[-1]='0'; m[n-2]=m[-2] = '1'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '1'+'0'+'1'='0'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '1'+'1'='0'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='1'; m[n-2]=m[-1] = '0'; estado "10".

Estado inicial "10", llega un '0'.

Entrada del codificador: m[n]=m[0]='0'.
En t_x, n=0, el estado es "10": m[n-1]=m[-1]='1'; m[n-2]=m[-2] = '0'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '0'+'1'+'0'='1'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '0'+'0'='0'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='0'; m[n-2]=m[-1] = '1'; estado "01".

Estado inicial "10", llega un '1'.

Entrada del codificador: m[n]=m[0]='1'.
En t_x, n=0, el estado es "10": m[n-1]=m[-1]='1'; m[n-2]=m[-2] = '0'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '1'+'1'+'0'='0'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '1'+'0'='1'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='1'; m[n-2]=m[-1] = '1'; estado "11".

Estado inicial "11", llega un '0'.

Entrada del codificador: m[n]=m[0]='0'.
En t_x, n=0, el estado es "11": m[n-1]=m[-1]='1'; m[n-2]=m[-2] = '1'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '0'+'1'+'1'='0'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '0'+'1'='1'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='0'; m[n-2]=m[-1] = '1'; estado "01".

Estado inicial "11", llega un '1'.

Entrada del codificador: m[n]=m[0]='1'.
En t_x, n=0, el estado es "11": m[n-1]=m[-1]='1'; m[n-2]=m[-2] = '1'.

Las salidas correspondientes para n=0 son:
c_0[n]=c_0[0]= m[n]+m[n-1]+m[n-2]=m[0]+m[-1]+m[-2] = '1'+'1'+'1'='1'
c_1[n]=c_0[0]= m[n]+m[n-2]=m[0]+m[-2] = '1'+'1'='0'

En t_{x+1}, correspondiente a n=1, tendremos el siguiente estado:
m[n-1]= m[0]='1'; m[n-2]=m[-1] = '1'; estado "11".

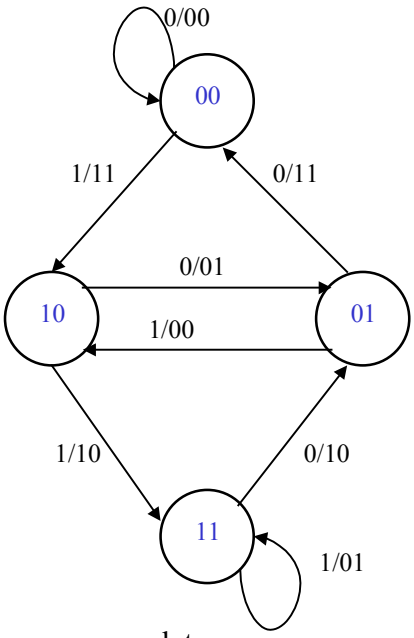
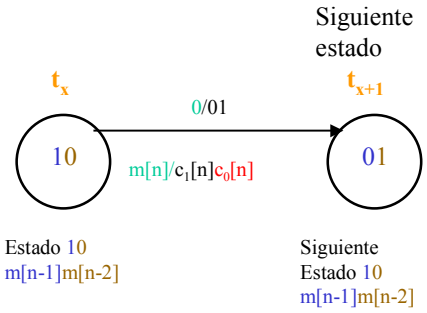
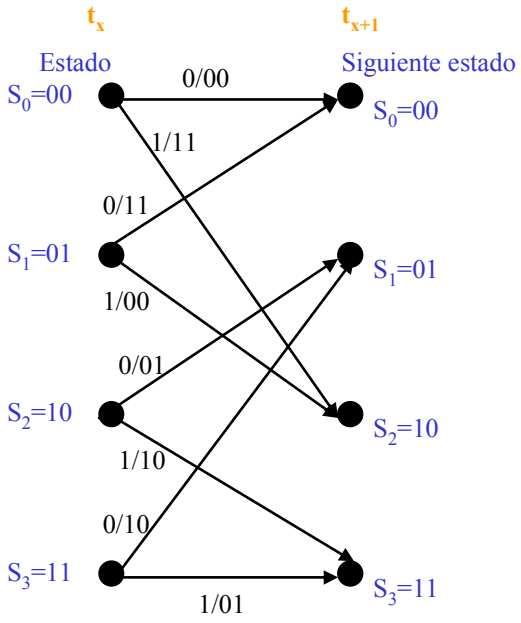


Diagrama completo

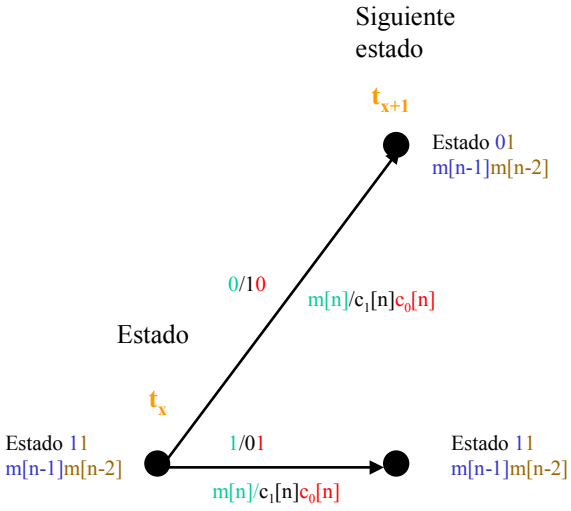


Notación empleada en las ramas y nodos

Figura 2.8: Diagrama de estados codificador ejemplo 1 (2, 1,K=3).



Malla trellis, una etapa



Notación empleada en las ramas y nodos

Figura 2.9: Malla trellis codificador ejemplo 1 (2, 1,K=3).

Tabla 2.8: Tabla estado/siguiente estado codificador ejemplo 1 (2, 1, K=3).						
Estado en t_x n=0 m[n-1]m[n-2]	Siguiente estado en t_{x+1}, n=1					
	Entrada m[0] = '0'			Entrada m[0] = '1'		
	Estado m[n-1]m[n-2]	Salidas		Estado m[n-1]m[n-2]	Salidas	
		$c_1[0]$	$c_0[0]$		$c_1[0]$	$c_0[0]$
00	00	0	0	10	1	1
01	00	1	1	10	0	0
10	01	0	1	11	1	0
11	01	1	0	11	0	1

2.5 Decodificación Viterbi. Fundamentos teóricos.

2.5.1 Objetivos.

En los *apartados 2.5 a 2.7* y en el *anexo C,* describimos de manera teórica y mediante un ejemplo como se realiza la decodificación.

En *2.5* nos centramos en: características, definiciones, aspectos teóricos y fórmulas más importantes. Para no extendernos demasiado no los detallamos de manera exhaustiva. No consideramos necesario hacerlo, porque son temas ampliamente tratados en la bibliografía: *2.8.1 B, C, D, E y F; 2.82, 2.83, 2.84 y 2.8.5*.

El funcionamiento y arquitectura lo describimos claramente en el *apartado 2.6*. Nos centramos en la parte teórica, porque los fundamentos y demostraciones matemáticas están desarrollados en el *anexo C* y en la bibliografía citada en *2.8*.

En *2.6* describimos como se realiza la decodificación paso a paso. Básicamente explicamos las siguientes acciones:

- Como añadir nuevas etapas a la malla trellis.
- Como actualizar el trellis.
- Como encontrar los caminos supervivientes y como actualizarlos.
- Como seleccionar cuál es el bit obtenido en la salida del decodificador en cada instante.

Por último, en *2.7* realizamos un ejemplo práctico, en el que se decodifica una secuencia codificada convolucionalmente. Este ejemplo resulta fundamental para consolidar los conocimientos del *apartado 2.6*.

El *apartado 2.7*, consiste en decodificar mediante Viterbi la secuencia de salida de un codificador convolucional: (2, 1, K=3), igual al del ejemplo 1 del *apartado 2.4.6*. La metodología para este decodificador, es la misma que la que se emplea en el Viterbi que implementamos en el proyecto: (2, 1, K=7). Por este motivo explicamos el proceso con un sistema sencillo de cuatro estados. Puesto que la explicación basándonos en el decodificador real con 64 estados sería extremadamente difícil. Además no aportaría ninguna ventaja, porque la metodología es la misma para los dos decodificadores. En las referencias *2.8.1.B* y *2.8.2*, también se explica la decodificación Viterbi empleando el mismo ejemplo con 4 estados.

Por el mismo motivo, el desarrollo del *apartado 2.6* y las *figuras: 2.10, y 2.15 a 2.23* también se basan en un decodificador asociado al codificador (2, 1, K=3). Se trata de la técnica habitual al describir la decodificación Viterbi. Por eso todas las referencias bibliográficas de *2.8* se basan en este ejemplo sencillo de 4 estados.

2.5.2 Codificación-decodificación, sistema FEC.

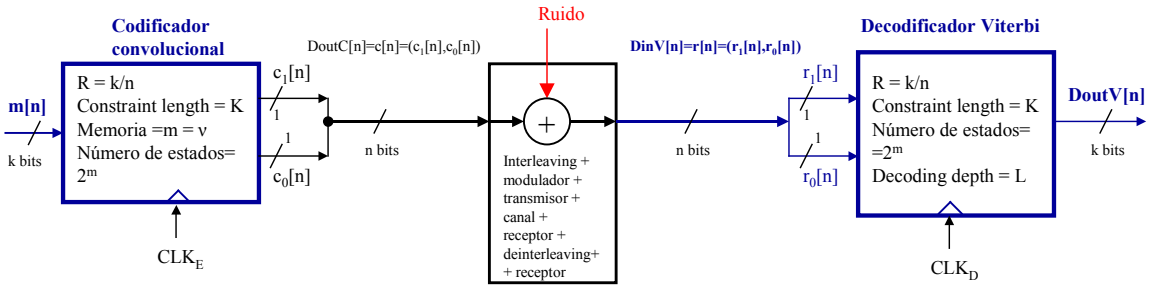


Figura 2.10: Decodificador Viterbi en un sistema FEC.

En un sistema de transmisión ideal, la secuencia recibida ideal: DinV[n], es exactamente igual a la transmitida retardada: c[n-retardo$_{canal}$]. De manera que no hace falta codificación-decodificación FEC.

Pero en un sistema real, las señales transmitidas sufren distintas perturbaciones y ruido, explicadas en *2.3.1*. Entonces se producen errores en los bits transmitidos, lo que hace que la secuencia recibida: DinV[n], sea diferente a la transmitida: c[n- retardo$_{canal}$].

Gracias a la codificación-decodificación FEC se consiguen disminuir los errores en la transmisión. De manera que la BER en la salida del decodificador será menor que la BER en la entrada. El rendimiento del decodificador se representa mediante gráficas BER en función de Eb/No, ejemplos en *apartado 7.7*.

Hay que tener en cuenta que la codificación añade redundancia, por lo que se transmiten más bits utilizando la misma potencia. Esto implica una energía por bit transmitido más baja, lo que repercute en un mayor número de errores. Por tanto, en un sistema con codificación convolucional y decodificación, se producen más errores en la transmisión que en un sistema sin codificar, porque la Eb/No es menor. Pero la ganancia del decodificador es mayor que la pérdida generada por la redundancia. Por lo que al final, con el sistema FEC, se consigue una BER más baja que un sistema sin codificación-decodificación. Las fórmulas que demuestran esto están en el *apartado 6.4.4*.

2.5.3 Características decodificador Viterbi.

Es el algoritmo más utilizado para decodificar un código convolucional. Se trata del decodificador óptimo, el de máxima verosimilitud: maximum likelihood sequence estimator, (MLSE). Pero también pueden emplearse otros decodificadores sub-óptimos, por ejemplo: el algoritmo de Fano, stack (también conocido como ZJ), y el M-algorithm. Estos algoritmos son útiles cuando la constraint length del decodificador es muy grande. Porque su rendimiento es inferior al de Viterbi, pero la complejidad computacional es menor. Se puede ampliar información sobre los decodificadores sub-óptimos en 510-522 de [1], 422-430 de [2] y 62-64 de [12].

En primer lugar detallamos las características que dependen directamente del codificador. Un decodificador está adaptado a un codificador convolucional determinado. De manera que si se cambian los parámetros del codificador, será necesario adaptar el algoritmo Viterbi a esos nuevos parámetros, que son los siguientes:

- Constraint length = K. Es la longitud del registro de desplazamiento más largo del codificador más uno. En el proyecto empleamos la definición c vista en *2.4.2*.

- Polinomio generador o función de transferencia del codificador.

- n = bits presentes en la entrada del decodificador en cada instante de tiempo. Al símbolo o dato de entrada de n bits lo denominamos: $DinV[n] = (r_{n-1}[n],...,r_0[n])$.

- k = Bits presentes en la salida del decodificador en cada instante de tiempo. Al símbolo o dato de salida de k bits lo denominamos: DoutV[n].

- Tasa o code rate $R_{code}= k/n$. Por cada n bits en la entrada del decodificador se obtienen k en la salida. La tasa o decode rate es el inverso: $R_{decode}=n/k$. Para evitar ambigüedades, en todo este documento siempre que nos refiramos a la tasa del decodificador, nos referiremos al code rate R=k/n. La cadena serie codificador-decodificador tiene tasa 1. Por cada k bits en la entrada del codificador se obtienen k bits en la salida del decodificador.

- Número de estados: $S = 2^{\nu} = 2^m$. Siendo $m = \nu$ el número de registros del codificador. Tanto el codificador como el decodificador tienen el mismo número de estados, que denominamos S. Siendo S_i el estado i.

Las siguientes características no dependen del codificador. De manera que no hay que cambiarlas aunque se modifique el codificador.

- En cada instante de tiempo llega un nuevo símbolo de entrada al decodificador: DinV[n]. En t_0, n=0, llega el primer dato: $DinV_1=DinV[0]$; en t_1, n=1, llega el segundo dato: $DinV_2=DinV[1]$; en n=2 el $DinV_3=DinV[2]$, y así sucesivamente. La manera genérica de representación es que en t_x, n=x, llega $DinV_{x+1}=DinV[x]$.

- Si la secuencia de entrada es finita: $DinV=\{DinV_1, DinV_2,\ldots,DinV_N\}=\{DinV[0], ...,DinV[N-1]\}$; el primer dato de entrada, $DinV_1$, llega en t_0, n=0; y el último dato $DinV_N$, llega en t_{N-1}, n=N-1.

- *Decoding depth* (profundidad de memoria). Puede denominarse como L ó Γ. Es el número de etapas que el decodificador guarda en la memoria antes de decodificar un símbolo de k bits. Cada vez que llega un dato $DinV_x$, se añade una nueva etapa al trellis. Y estas etapas de trellis se almacenan en la memoria del decodificador.

- Para que el decodificador tenga el rendimiento adecuado, es necesario $L= \Gamma \geq 5K$. Si se utiliza $L<5K$, entonces el rendimiento empeora de manera significativa respecto al que se obtendría con $L\geq5K$. Se pueden utilizar valores mayores de 5K, pero la mejora de rendimiento es muy pequeña respecto a la que se obtiene con L=5K. Todos estos fundamentos teóricos los demostramos utilizando el decodificador que hemos implementado, en el *apartado 7.7.3*.

- El decodificador almacena decoding depth etapas de trellis en su memoria. Estas etapas abarcan desde la primera posición, correspondiente al instante $t_{x-(L-1)}$, hasta la última posición del trellis en t_x.

- La memoria es fija con L posiciones. Entonces entre $0 \leq n=x\leq L-1$, la memoria aún no está completa. En este caso, en cada instante de tiempo se almacena una nueva etapa del trellis. La primera posición corresponde a t_0 y la última a t_x, con $x \leq L-1$.

- Para $L \leq n=x \leq N-1$, al añadir una nueva etapa a la memoria hay que eliminar otra, porque la memoria ya está completa. En t_x, la primera posición corresponde a $t_{x-(L-1)}$, y la última a t_x. Al llegar un nuevo dato $DinV_{x+1}$ se añade una etapa al trellis, y las etapas almacenadas en la memoria se desplazan una posición. De manera que la primera pasa a ser la $t_{x-(L-2)}$ y la última la t_{x+1}. Por tanto la estructura es FIFO, porque al añadir una nueva etapa, de t_x a t_{x+1}, sale de la memoria la etapa t_{x-L} a $t_{x-(L-1)}$.

- La latencia del decodificador ideal es L +1. En t_0 llega el primer dato $DinV_1$ al decodificador, pero en la salida no se obtiene el primer dato válido hasta t_{L+1}. En t_{L+1} se obtiene DoutV[n=L+1], que corresponde a la decodificación de $DinV_1$. En un instante genérico, t_x, n= x, se obtiene DoutV[n=x] que corresponde a la decodificación de $DinV_{x-L}$.

- El período de proceso de un dato es el tiempo que necesita el decodificador entre dos símbolos de entrada consecutivos. Al decodificador llega un símbolo de entrada por cada período de proceso, y se obtiene un dato en la salida en cada período de proceso. Por tanto, el período de proceso es el tiempo que transcurre entre un instante t_x y el siguiente t_{x+1}, o con otra nomenclatura, entre n= x y n = x+1.

- Dependiendo de la arquitectura, el período de proceso coincidirá con el período de reloj del decodificador, T_{CLKD}, o no. En la arquitectura totalmente paralelo, el decodificador realiza las acciones necesarias en el intervalo entre t_x y t_{x+1}, en un sólo T_{CLKD}. Por lo que el período de proceso de un dato en el decodificador es igual a T_{CLKD}. En este caso, la frecuencia de reloj del codificador (F_{CLKE}) es igual a la del decodificador (F_{CLKD}).

- Sin embargo en una estructura serie, la decodificación se realiza en varias etapas, y cada etapa requiere un tiempo T_{CLKD} para procesarse. De manera que el intervalo t_x a t_{x+1} requiere un tiempo: NumEtapas*T_{CLKD}. Entonces, el período de proceso de un dato en el decodificador es: NumEtapas*T_{CLKD}. En este caso la frecuencia de reloj del codificador y del decodificador no serán iguales. El decodificador trabaja con F_{CLKD}, pero la frecuencia con la que llegan datos DinV[n] en su entrada debe ser F_{CLKD}/NumEtapas. El codificador trabaja con F_{CLKE} y obtiene un dato en su salida cada T_{CLKE} segundos. De manera que se necesita $F_{CLKE} = F_{CLKD}$/NumEtapas.

- Puede emplearse *hard decoding* o *soft decoding*, (decodificación dura o blanda). La decodificación dura consiste en que la señal recibida en el receptor se cuantifica con 2 niveles, '0' ó '1', por lo que se emplea una cuantificación de 1 bit. En decodificación *soft* se emplea más de un bit en la cuantificación, por lo que el receptor decide entre $2^{NumBits}$ niveles de cuantificación, ver siguiente figura:

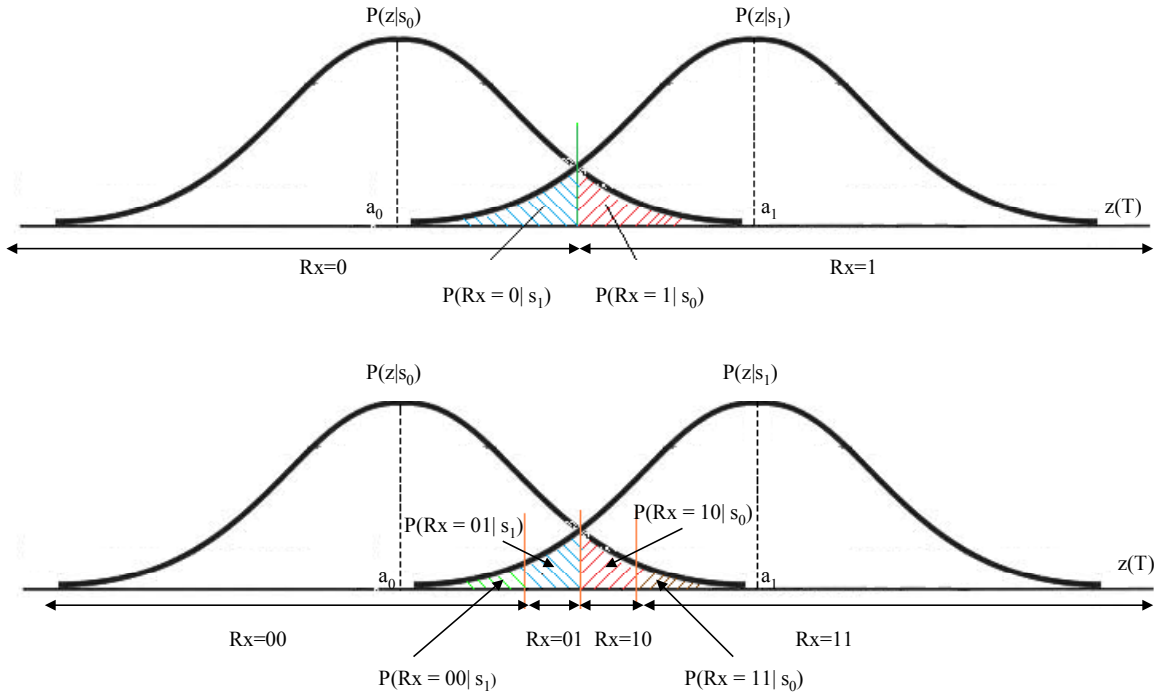


Figura 2.11: Cuantificación empleando 1 y 2 bits.

Nota de la figura: Z(T) es la energía recibida en el receptor. El transmisor transmite dos símbolos: s_0 y s_1. El receptor decide '0' ó '1' si se emplea un bit de cuantificación, y "00", "01", "10" y "11" si se emplean dos bits de cuantificación.

- La decodificación *soft* usando 8 niveles de cuantificación, 3 bits, consigue una ganancia de 2 dB respecto a la decodificación dura. Utilizar más de 3 bits en la cuantificación no aumenta la ganancia de manera significativa. En nuestro proyecto implementamos un decodificador duro, porque es el que se nos exige en las especificaciones. Por este motivo no detallamos los aspectos de la decodificación soft. Pero incluimos una amplia bibliografía sobre decodificación soft en *2.8.1F*.

- Cuando los datos de entrada se organizan en tramas finitas de N símbolos: $DinV[n]=\{DinV_1,DinV_2.....,DinV_N\}$, hay 3 formas de realizar la decodificación:

    - *Trellis truncation* (truncamiento de trellis): La trama consta de N símbolos. Cuando llega el primer dato, se empiezan a formar los caminos en el trellis partiendo del estado cero. Cuando llega el último dato, no se hace nada para forzar a que el decodificador acabe en un estado determinado. Esto implica que siempre es necesario resetear al decodificador antes del comienzo de la trama de entrada, para que se sitúe en el momento inicial en el estado 0. La ventaja de esta técnica es que es la más sencilla. El inconveniente es que los últimos decoding depth símbolos de la trama tienen menos protección contra errores. Porque se utilizan menos de L posiciones de memoria en su decodificación.

    - *Trellis termination* o *zero tail* (terminación del trellis o cola de ceros): En este caso la trama consta de N+m símbolos: $DinV[n]=\{DinV_1,DinV_2.....,DinV_{N+m}\}$. El decodificador parte del estado cero cuando llega $DinV_1$, y finaliza en el estado cero cuando llega $DinV_{N+m}$. Con este sistema, los últimos bits de la trama de entrada tienen la misma protección frente a errores que el resto de bits de la trama. Por tanto elimina la desventaja del truncamiento de trellis. El inconveniente es que para forzar a que el estado final sea cero, es necesario añadir una cola, (*tail*), de m símbolos de valor cero al final de cada trama de entrada al codificador. De manera que se transmiten N símbolos de datos + m redundantes. Otro inconveniente frente a truncamiento de trellis es que el decodificador es más complejo.

    - *Tail biting*: En esta técnica no hay que añadir símbolos redundantes en la entrada del codificador, de manera que la secuencia de entrada y la transmitida tendrán N símbolos. El estado del codificador es el mismo tanto al inicio como al final de la trama. Esto se consigue utilizando los últimos m símbolos de la trama de entrada al codificador para inicializar su estado. El mayor inconveniente, es que antes de transmitir el primer símbolo codificado, es necesario que el último dato, m_N, esté disponible en la entrada al codificador, por tanto la latencia aumenta. Además la complejidad del decodificador también aumenta.

En nuestro proyecto la especificación del decodificador es la de truncamiento de trellis. Por eso en todo este documento y en el proyecto nos referiremos siempre a esta técnica.

No entramos en más detalle describiendo las técnicas porque ya lo hemos hecho en este mismo documento, *apartado 7.7.4*. También se puede ampliar información en [44] y páginas 138-141 de [16].

2.6 Arquitectura y funcionamiento decodificador Viterbi.

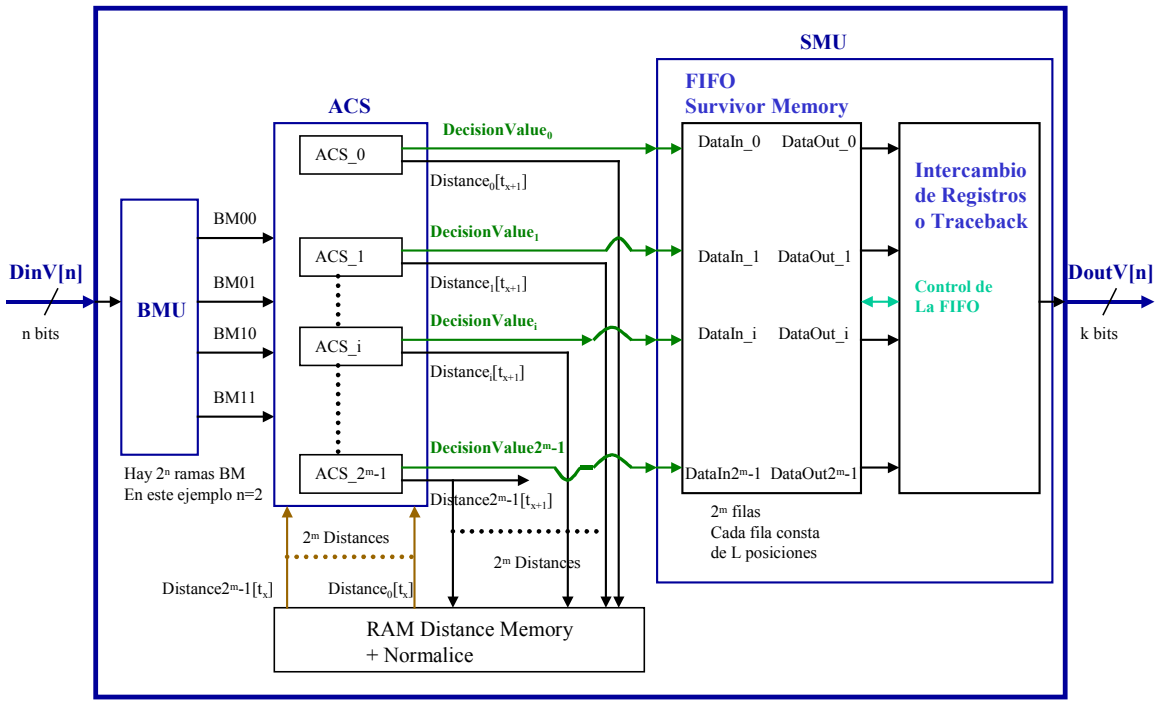


Figura 2.12: Arquitectura genérica decodificador Viterbi.

A continuación exponemos brevemente algunos aspectos de la arquitectura. La descripción completa tanto de la arquitectura como del funcionamiento la realizamos en los *apartados: 2.6.1, 2.6.2 y 2.6.3*. También puede ampliarse información en la bibliografía: *2.8.1 C y D; 2.8.3, 2.8.4 y 2.8.5*. Por último, el *apartado 2.7* es una ayuda fundamental para la comprensión del funcionamiento, porque es un ejemplo práctico de todo el desarrollo teórico.

El bloque SMU, Survivor Memory Unit, se encarga de almacenar y actualizar los caminos supervivientes. También selecciona el camino superviviente ganador y obtiene a partir de él el dato decodificado final, DoutV, que corresponde a la primera rama del camino superviviente ganador. Este bloque puede realizarse de varias maneras diferentes, pero las más comunes son dos: método intercambio de registros y *traceback*. En nuestro proyecto implementamos el intercambio de registros y por eso lo describimos ampliamente en *2.6.1, 2.6.2* y *2.6.3*.

El método *traceback* no es de interés para nuestro proyecto, por eso sólo lo describimos brevemente en *2.6.4*. Además en ese apartado compararemos ambos métodos.

La decodificación puede implementarse mediante las tres técnicas descritas en *2.5.3*:

1) Truncamiento de trellis.
2) Cola de ceros.
3) *Tail biting*.

En nuestro proyecto empleamos truncamiento de trellis, por eso, para no alargar innecesariamente este documento, describimos únicamente este método. Los otros dos pueden consultarse en [44] y páginas 138-141 de [16].

La tarea del decodificador consiste en que en un período de proceso, que abarca desde t_x a t_{x+1}, (o con otra nomenclatura entre n = x y n = x+1), debe realizar estas acciones:

1. En t_x están almacenadas todas las distancias hasta cada estado i en t_x.

2. En t_x están almacenados todos los caminos supervivientes hasta cada estado i en t_x. La definición de distancias y caminos supervivientes se verá en *2.6.1*.

3. Entre t_x y t_{x+1} el decodificador debe calcular las distancias hasta cada estado i en t_{x+1}. Y también los caminos supervivientes hasta cada estado i en t_{x+1}.

4. El proceso se repite, de manera que siempre en cada período de proceso se parte de las distancias y caminos supervivientes en t_x y se obtienen los de t_{x+1}.

5. Cada vez que llega un nuevo dato al decodificador, se añade una nueva etapa al trellis. En t_x llega $DinV_{x+1}$, y en t_{x+1} ya se habrá añadido una nueva etapa al trellis, y se habrán calculado las distancias y caminos resultantes de añadir $DinV_{x+1}$ al trellis. Entonces decimos que en t_x llega $DinV_{x+1}$ y ya está situado en el trellis $DinV_x$. En t_{x+1} llega $DinV_{x+2}$ y ya está situado en el trellis $DinV_{x+1}$. Y así sucesivamente...

6. En cada período de proceso llega un dato. En t_0 llega el primero, $DinV_1$ y en t_{N-1} el último, $DinV_N$.

7. Una vez superada la latencia de decoding depth = L = Γ períodos de proceso, el decodificador obtiene un dato decodificado en cada período de proceso. Entonces el primer dato $DoutV_1$ se obtiene en t_{L+1}, y corresponde a la decodificación de $DinV_1$. El último $DoutV_N$ en t_{N+L} y corresponde a la decodificación de $DinV_N$. En un instante genérico t_x se obtiene $DoutV_{x-L}$=DoutV[n=x], que corresponde a la decodificación de $DinV_{x-L}$.

Atendiendo al proceso anterior, la decodificación se divide en tres etapas:

1) Inicio de la trama de entrada. Es el intervalo que transcurre entre t_0 y t_{L-1}, intervalo en el que llegan los datos $DinV_1$....$DinV_L$. En este intervalo de tiempo, no se obtienen bits decodificados en la salida del decodificador. Porque la memoria FIFO que almacena los caminos supervivientes aún no está completa.

2) Caso general. Es el intervalo que transcurre entre t_L y t_{N-1}. Momento en el que llegan los símbolos $DinV_{L+1}$ a $DinV_N$. En este tiempo en cada período de proceso se obtiene un dato decodificado DoutV[t_x]. La característica fundamental de este estado es que la inicialización ha terminado, de manera que la memoria FIFO con los caminos supervivientes está completa. En cada período de proceso, entre t_x y t_{x+1}, llega un nuevo dato $DinV_{x+1}$ al decodificador y se introduce en la malla trellis, añadiendo una etapa más a la malla. Pero el tamaño de la memoria es fijo, sólo se pueden almacenar L etapas de trellis. Así que al añadir una nueva etapa, la que pasa de t_x a t_{x+1}, se debe eliminar la más antigua, la que transcurre entre t_{x-L} y $t_{x-(L-1)}$.

3) Final de la trama de entrada. En t_{N-1} llega el último símbolo: $DinV_N$. A partir de ese momento el decodificador continúa trabajando, puesto que tiene que decodificar las L etapas que aún tiene almacenadas en su memoria. La velocidad de decodificación se mantiene, de manera que se sigue obteniendo un símbolo en DoutV[n] en cada instante de tiempo. Entonces, en t_{N-1} se obtiene la decodificación correspondiente a $DinV_{N-(L+1)}$. En t_N la correspondiente a $DinV_{N-L}$; en t_{N+1} la de $DinV_{N-(L-1)}$ … Y así sucesivamente hasta que en t_{N+L}, se obtiene la decodificación del último símbolo de entrada $DinV_N$.

En los *apartados 2.6.1, 2.6.2 y 2.6.3* detallamos todo el proceso de decodificación de manera teórica. La lectura de los 3 puntos puede combinarse con la del *apartado 2.7*, lo que facilita la comprensión del proceso.

2.6.1 Caso general empleando intercambio de registros.

Decidimos describir este caso, pese a que no es el que primero transcurre en el tiempo, porque es el más genérico. Los casos inicial y final no son más que un caso particular, de manera que resulta mucho más claro desarrollarlos en los siguientes apartados.

La decodificación Viterbi se basa en ir formando caminos en una malla trellis, a partir de la secuencia DinV[n] recibida. Un camino es una secuencia de ramas que comienzan en el estado inicial, cero en t_0, y finalizan en un estado o nodo i en un instante de tiempo t_x, ver *figura 2.15*.

La característica fundamental de un camino es su distancia, peso o path metric. **$Distance_i[t_x]$= $Distance_i[x]$** es la distancia hasta el estado i en t_x, n=x. La distancia es igual a la suma de los pesos de cada una de las ramas, desde el origen en t_0, hasta el final en t_x, ver *figura 2.13*.

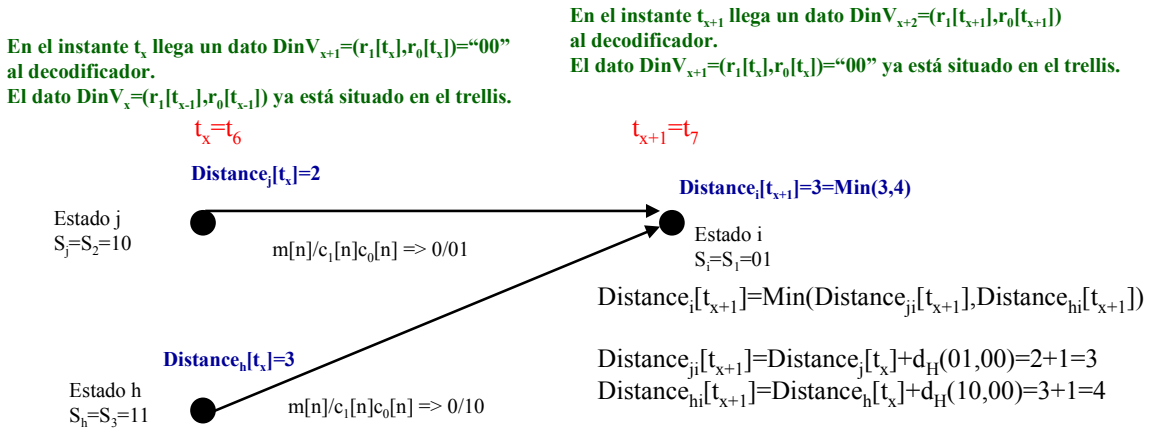


Figura 2.13: Cálculo de $Distance_i[t_{x+1}]$

$$d_H(x,y)=\sum_{i=1}^{n}[x_i \neq y_i]; \qquad \text{siendo } [x_i \neq y_i]=\begin{cases}1 & \text{si } x_i \neq y_i \\ 0 & \text{si } x_i = y_i\end{cases}$$

En la *figura 2.13* se aprecia como se calculan las distancias, los pasos son los siguientes:

1. La distancia hasta un estado i en t_{x+1} es igual a la suma de los pesos de cada una de las ramas desde el origen en t_0, hasta el final en t_{x+1}. Sin embargo esta operación es muy costosa computacionalmente, por ese motivo el decodificador emplea otro método más simple, pero igual de exacto. Se basa en que todo estado i en t_{x+1} tiene 2^k estados anteriores en t_x. En nuestro ejemplo k=1, de manera que tiene dos estados anteriores, que denominamos j y h. Entonces la distancia hasta i en t_{x+1}, es igual a la distancia hasta el estado anterior, j ó h en t_x, más el peso de la rama que transiciona desde j ó h en t_x, a i en t_{x+1}. El peso de la rama, o branch metric, es igual a la distancia de Hamming, d_H, entre el símbolo $c_1[n]c_0[n]$ que caracteriza la rama y el símbolo DinV recibido en t_x.

   Peso rama que transiciona del estado j en t_x a i en t_{x+1} = $Peso_{ji}[t_{x\rightarrow x+1}]$=

$$d_H(DoutC[t_x], DinV[t_x]) = d_H(c_1c_0, r_1r_0) = \sum_{i=0}^{1}[c_i \neq r_i];\ \text{siendo}\ [c_i \neq r_i] = \begin{cases} 1 & \text{si } c_i \neq r_i \\ 0 & \text{si } c_i = r_i \end{cases}$$

2. Como cada estado tiene 2^k anteriores, habrá 2^k distancias, que denominamos:

   - $Distance_{ji}[t_{x+1}]$ es la distancia hasta llegar hasta el estado i en t_{x+1}, cuando el estado anterior, en t_x, es el j.
   - $Distance_{hi}[t_{x+1}]$ es la distancia hasta llegar hasta el estado i en t_{x+1}, cuando el estado anterior, en t_x, es el h.
   - El decodificador Viterbi siempre selecciona la menor de ellas, y la otra la desecha. Siempre hay que seleccionar la menor distancia, porque como se demuestra en el *anexo C*, es la que maximiza la verosimilitud. Entonces la distancia hasta un estado i en t_{x+1}, n=x+1 es igual a:

$$Distance_i[t_{x+1}] = Min(Distance_{ji}[t_{x+1}], Distance_{hi}[t_{x+1}]).$$

El cálculo de las distancias se realiza mediante los bloques *BMU* y *ACS*.

Bloque BMU, Branch Metric Unit. (Más información en *2.8.1C, 2.8.3* y *2.8.4*).

Se encarga de calcular la distancia de Hamming entre el símbolo recibido $r_1[t_x]r_0[t_x]$ y el símbolo $c_1[n]c_0[n]$ que caracteriza la rama. De manera que calcula $Peso_{ji}[t_{x\rightarrow x+1}]$, que veíamos anteriormente. En el ejemplo de la *figura 2.12*, el símbolo recibido tiene 2 bits, n=2, por eso en cada período de proceso, este bloque debe realizar cuatro operaciones y obtendrá:

1. $BM00=d_H("00", r_1[t_x]r_0[t_x])$
2. $BM01=d_H("01", r_1[t_x]r_0[t_x])$
3. $BM10=d_H("10", r_1[t_x]r_0[t_x])$
4. $BM11=d_H("00", r_1[t_x]r_0[t_x])$
5. Una rama BMU puede tener un valor = 0, 1 ó 2. Por tanto se necesitan dos bits para codificar cada rama.

Bloque ACS, Add Compare Select. (Más información en *2.8.1C, 2.8.3* y *2.8.4*).

Cada bloque *ACS* obtiene la distancia hasta un estado i en t_{x+1}, $Distance_i[t_{x+1}]$. En nuestro ejemplo, k=1, de manera que realiza estas operaciones:

- $Distance_{ji}[t_{x+1}] = Distance_j[t_x] + Peso_{ji}[t_{x\rightarrow x+1}]$. Operación add.
- $Distance_{hi}[t_{x+1}] = Distance_h[t_x] + Peso_{hi}[t_{x\rightarrow x+1}]$. Operación add.
- $Distance_i[t_{x+1}] = Min(Distance_{ji}[t_{x+1}], Distance_{hi}[t_{x+1}])$. Operación compare.
- La operación select consiste en seleccionar cuál es la rama ganadora, de entre las 2^k ramas que llegan hasta el estado i en t_{x+1}. En nuestro ejemplo debe decidir entre la rama ji y la rama hi. Siempre elegirá la rama con la que se minimiza $Distance_i[t_{x+1}]$:
  - Si $Min(Distance_{ji}[t_{x+1}], Distance_{hi}[t_{x+1}]) = Distance_{ji}[t_{x+1}]$, entonces se selecciona como ganadora la rama ji. Se dice que el estado ganador anterior a i en t_{x+1} es el j en t_x. $DecisionValue[t_{x+1}] = j$.
  - Si $Min(Distance_{ji}[t_{x+1}], Distance_{hi}[t_{x+1}]) = Distance_{hi}[t_{x+1}]$, entonces se selecciona como ganadora la rama hi. Se dice que el estado ganador anterior a i en t_{x+1} es el h en t_x. $DecisionValue[t_{x+1}] = h$.
  - Si $Distance_{ji}[t_{x+1}] = Distance_{hi}[t_{x+1}]$, entonces puede elegirse de manera aleatoria, cualquiera de las dos ramas como ganadora.

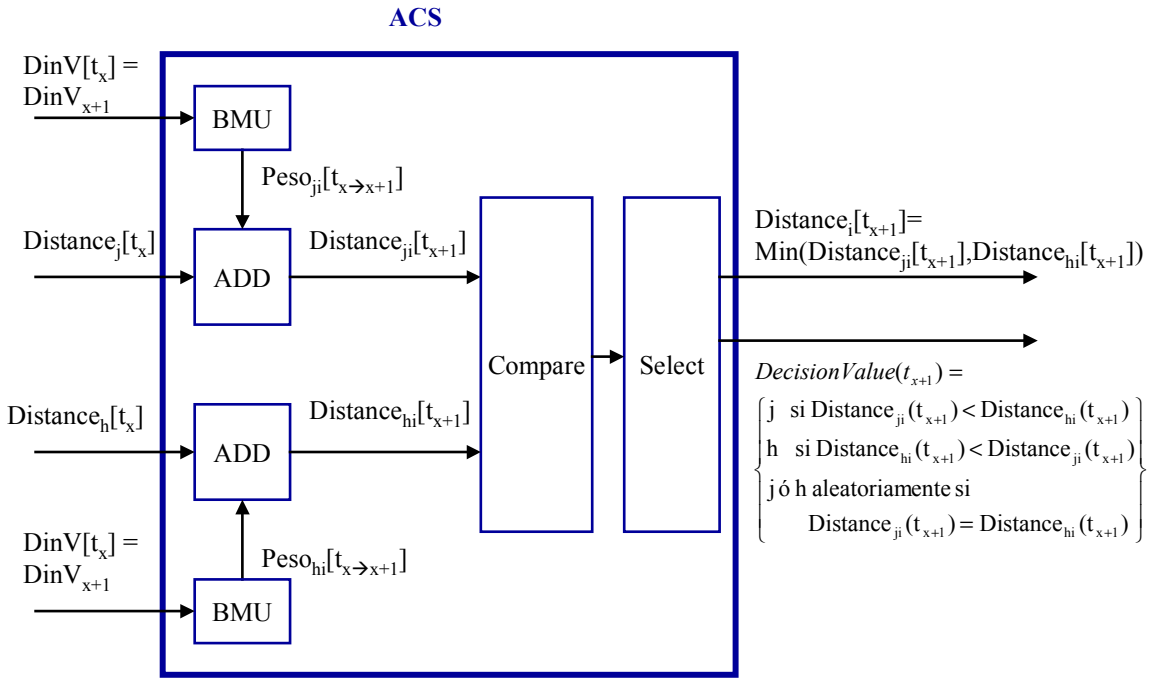


Figura 2.14: Arquitectura módulo ACS.

Se necesitan un total de 2^m-1 bloques ACS, uno para cada estado. Estos bloques pueden organizarse en serie, en paralelo o una combinación de ambos. Lógicamente, la estructura serie es la que menos área ocupa, pero la que maximiza el período de proceso de un dato. Porque cada etapa serie precisa de un T_{CLK}, de manera que el período de proceso de un dato es igual a: T_{CLK}*(número de etapas serie). Sin embargo, en la estructura paralelo, todas las operaciones se realizan en un sólo T_{CLK}. En este caso el período de proceso es el mínimo, e igual a un T_{CLK}.

En nuestro proyecto utilizamos una combinación de 8 etapas serie y 8 etapas en paralelo. De esta manera optimizamos el compromiso entre área y velocidad.

En el período de proceso entre t_x y t_{x+1} el decodificador debe calcular y almacenar en una memoria RAM todos los valores de $Distance_i[t_{x+1}]$, para todo i ∈ [0..2^m-1]. Esta tarea la realiza mediante los bloques BMU y ACS. Además el proceso es cíclico, porque para obtener los valores correspondientes a t_{x+1}, se necesitan las distancias previas correspondientes a t_x: $Distance_i[t_x]$, para todo i ∈ [0..2^m-1]. Por tanto se necesitan dos memorias RAM. Al inicio del período de proceso, una de las memorias tiene almacenadas las distancias en t_x, que se utilizan como referencia. En la otra se van almacenando los nuevos valores, correspondientes a t_{x+1}. De esta manera, en el siguiente período entre t_{x+1} y t_{x+2}, una memoria contendrá las distancias de referencia, correspondientes a t_{x+1}, y en la otra se irán almacenando las correspondientes a t_{x+2}.

Normalización.

Con el transcurrir de las etapas del trellis, la distancia hasta llegar a cada uno de los nodos va aumentando. Por tanto existe la posibilidad de desbordamiento. Para evitarlo hay que normalizar las distancias, en nuestro proyecto lo realizamos de esta manera:

- Se define una constante, NormalizeTime.

- Cada NormalizeTime etapas de trellis se calcula la distancia mínima de entre las 2^m distancias.
  $$MinDistance[n = NormaliceTime] = \min_{0 \le i \le 2^m - 1} \left(Distance_i[n = NormaliceTime] \right).$$

- Cada NormalizeTime etapas se realiza la normalización. Para ello a todas las distancias se les resta el valor de distancia mínima:
  $Distance_i$=$Distance_i$-MinDistance; se realiza para todo i ∈ [0..2^m-1].

Otra opción habitual consiste en realizar la normalización cuando alguna de las distancias supera un valor determinado.

Al normalizar, los pesos ya no representan el valor real, sino un valor relativo. Pero esto no disminuye nada el rendimiento del decodificador. Porque en el proceso de decodificación en ningún momento se necesita el valor absoluto real de los pesos. Únicamente se necesita compararlos para determinar cuál es el menor de ellos, que constituye el estado ganador. Y esta acción se puede realizar perfectamente con los valores relativos.

Se puede ampliar información sobre normalización en [45], [46] y [47].

Bloque SMU, Survivor Memory Unit (unidad de memoria superviviente).

(Más información en *2.8.1D, 2.8.3* y *2.8.5*).

El bloque se encarga de almacenar y actualizar los caminos supervivientes, seleccionar el camino superviviente ganador y obtener el dato decodificado final, DoutV, que corresponde al primer bit del camino superviviente ganador. Para describir el funcionamiento, en primer lugar debemos definir lo que es un camino.

$Camino_i(0:(L-1))[t_x] = Camino_i(0:(L-1))[x]$: Es un vector con L posiciones, que representa al camino hasta el estado i en t_x, n=x. Constituye la secuencia de ramas de trellis que hay que seguir para llegar hasta el nodo i en el instante t_x. Sus características son:

a) Cada camino consta de decoding depth = L = Γ ramas o posiciones.

b) La casilla 0 corresponde al origen del camino, primer bit, en $t_{x-(L-1)}$.

c) La casilla L-1 corresponde al final del camino, último bit, en t_x.

d) La secuencia de ramas se representa como una secuencia de bits. Cada rama está caracterizada por $m[n]/c_1[n]c_0[n]$. De manera que el camino hasta el estado i en t_x, es la secuencia de bits m[n] que hay que seguir para desplazarse desde la primera posición del trellis, instante $t_{x-(L-1)}$, hasta la última, t_x. Ver *figuras 2.15, 2.16 y 2.17*.

e) Por ejemplo en la *figura 2.15*, $camino_1(0:4)[6]$ = "10010", representa el camino hasta llegar al estado 1 en t_6. En la secuencia el bit más a la izquierda siempre corresponde a la primera posición, la 0. De manera que $camino_1(0)[t_6]$='1' es el primer bit del camino, y $camino_1(4)[6]$='0' es el último bit.

f) *Nota 2.1*: En el *tema 5* los caminos se definen como $camino(i)(4:0)[t_x]$. Así que en ese *tema 5,* la secuencia se escribe en orden inverso y sería $camino(1)(4:0)[t_6]$="01001". Este cambio es imprescindible, porque la representación $camino_1(0:4)[t_6]$ es más intuitiva, pero no podemos realizarla en hardware, porque System Generator exige que la indexación de los vectores sea siempre con downto.

g) Cada camino se caracteriza por $Distance_i[t_x]$, que obtiene el bloque ACS.

Camino superviviente. Al describir el módulo ACS vimos que hasta cada estado i llegan 2^k caminos. Pero sólo interesa uno de ellos, el que minimiza $Distance_i[x+1]$. En el ejemplo de 4 estados y en nuestro proyecto k siempre es 1. Así que al añadir una nueva etapa al trellis siempre hay dos caminos posibles hasta i: el que tiene como estado anterior el j y el que tiene el h. Entonces:

a) Si $DecisionValue[t_{x+1}]=$ j, el camino superviviente hasta i en t_{x+1} es el que tiene como estado anterior el j.

b) Si $DecisionValue[t_{x+1}]=$ h, el camino superviviente hasta i en t_{x+1} es el que tiene como estado anterior el h.

Entonces en t_{x+1} hay 2^{m+1} caminos, dos hasta cada uno de los estados. Pero sólo se almacenan en la memoria los supervivientes, el resto se desechan. Hay 2^m caminos supervivientes, uno hasta cada estado i, con i $\in [0..2^m-1]$.

Cada camino consta de L casillas y cada casilla contiene k bits. Por tanto el decodificador necesita una memoria de 2^m *kL bits. 2^m filas por kL bits en cada fila.

Camino superviviente ganador: Es aquel, de entre los 2^m caminos supervivientes almacenados en la memoria, que tiene una menor distancia en su última posición.

La manera más inmediata para comprender el funcionamiento del decodificador se expresa en las siguientes 3 figuras y la *tabla 2.9*. En ellas se representa todo el trabajo del decodificador en un período de proceso que hemos tomado como ejemplo, entre t_6 y t_7. En la *figura 2.15* se describe la situación del decodificador al inicio del período de proceso, instante t_6. A continuación, en *2.16*, el trabajo que se realiza durante el período de proceso, entre t_6 y t_7. Por último, en *2.17*, la situación del decodificador al finalizar el período de proceso, instante t_7.

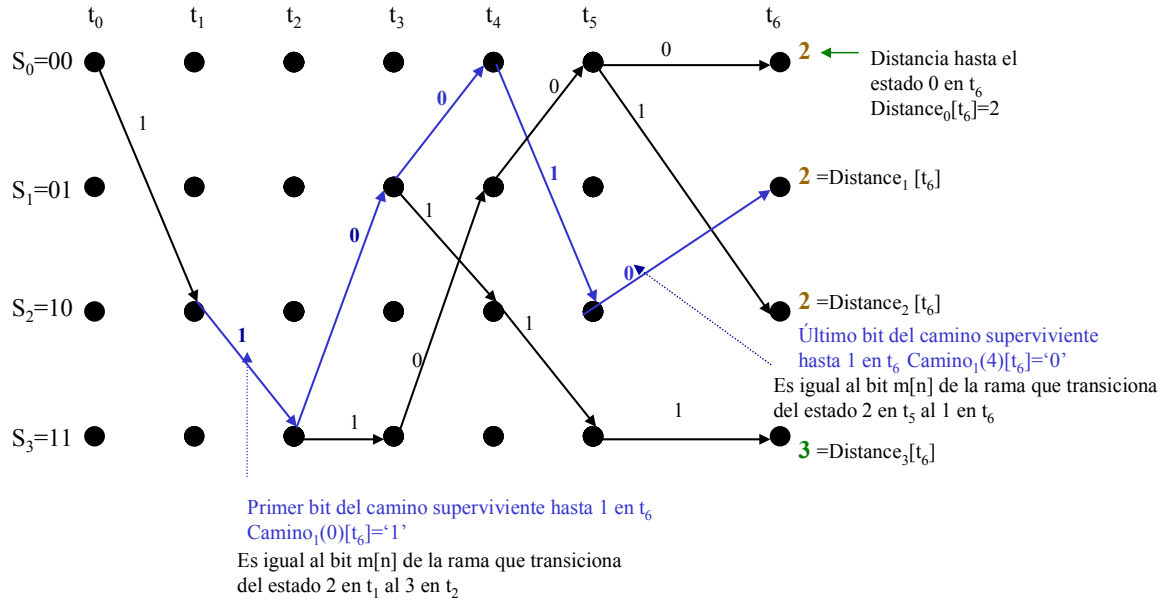


Figura 2.15: Situación de los caminos en t_6, al inicio del período de proceso.

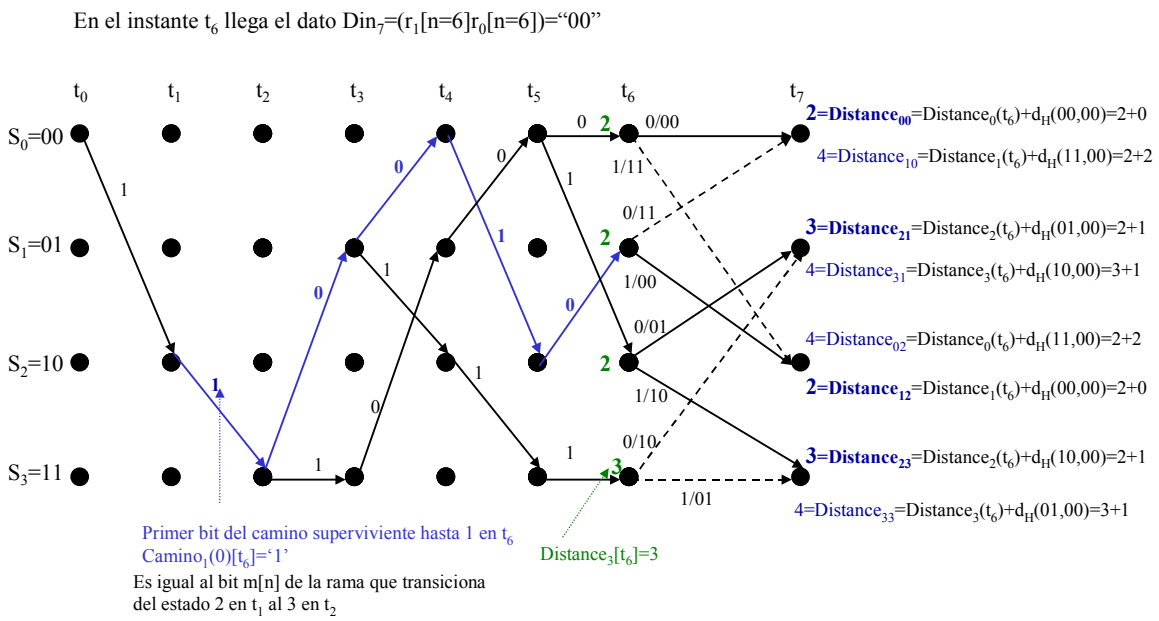


Figura 2.16: Trabajo del decodificador en el período de proceso entre t_6 y t_7.

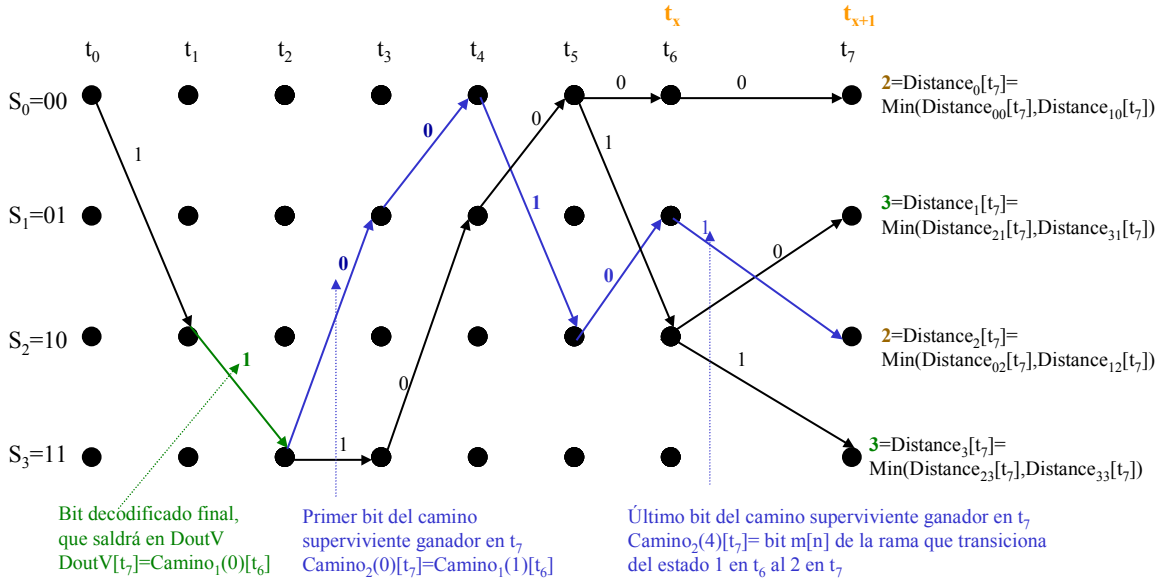


Figura 2.17: Situación de los caminos en t_7, final del período de proceso.

Tabla 2.9: Caminos supervivientes almacenados en la memoria en t_6 y t_7.		
	t_6, n=6	t_7, n=7
DinV[n]=$r_1[n]r_0[n]$	DinV[6]=DinV$_7$= 00	DinV[7]=DinV$_8$= 01
Camino$_0$(0:4)[n]	11000	10000
Camino$_1$(0:4)[n]	**10010** (superviviente ganador)	10010
Camino$_2$(0:4)[n]	11001	**00101** (ganador)
Camino$_3$(0:4)[n]	10111	10011
DoutV[n]	Para obtenerlo se necesitan los caminos en n=5	**1**=Camino$_1$(0)[6]

En todas las figuras y tablas de este tema que contengan etapas de malla trellis, utilizamos este código de colores: el camino superviviente ganador en azul, el bit seleccionado para DoutV[n] en verde y la distancia mínima en marrón.

Tras las definiciones ya se puede detallar paso a paso el funcionamiento del SMU:

Estamos tratando el caso general, cuya característica fundamental es que la inicialización ha terminado, de manera que la malla trellis está completa. En cada período de proceso llega un nuevo dato DinV[n] al decodificador y se introduce en el trellis, añadiendo una etapa más a la malla. Pero el tamaño de la memoria es fijo, sólo se pueden almacenar L etapas de trellis. Entonces al añadir una nueva etapa, la que transiciona entre t_x y t_{x+1}, se debe eliminar la más antigua, la que transcurre entre t_{x-L} y $t_{x-(L1)}$.

A continuación describimos las acciones que se deben realizar en un período de proceso genérico, entre t_x y t_{x+1}. Hay que realizar 2 acciones fundamentales:

A. En t_x los caminos supervivientes están almacenados en la memoria del decodificador, FIFO survivor memory, y su última posición corresponde a t_x. Entonces hay que actualizar los caminos para que cuando llegue el siguiente instante, t_{x+1}, los caminos estén actualizados y su última posición corresponda a t_{x+1}.

B. Es necesario obtener el dato, k bits, decodificado final en la salida del decodificador en el instante t_{x+1}. DoutV[t_{x+1}] = DoutV[n=x+1].

Los pasos que permiten realizar las 2 acciones anteriores son:

1. En cada instante de tiempo, t_x, llega un nuevo símbolo, $DinV_{x+1}$, al decodificador. Por tanto hay que añadir una nueva etapa al trellis. Esta etapa tendrá su origen en t_x y su final en t_{x+1}.

2. Al añadir una nueva etapa al trellis, es necesario eliminar otra, porque el número de etapas almacenadas en la memoria es fijo, e igual a L. Este proceso es:

    - En t_x el trellis tiene su primera posición en $t_{x-(L-1)}$ y la última en t_x.
    - Se añade una nueva etapa, que abarca de t_x a t_{x+1}.
    - La etapa t_{x-L} a $t_{x-(L-1)}$ sale de la memoria del decodificador.
    - En el instante t_{x+1} la primera posición del trellis corresponderá a $t_{x-(L-2)}$ y la última a t_{x+1}.

3. El bloque SMU tiene 2 entradas para cada estado i:

    a) $Distance_i[t_{x+1}]$.

    b) El estado seleccionado o ganador, j ó h en t_x, anterior a cada estado i en t_{x+1}. $DecisionValue_i[t_{x+1}]$= j ó h.

4. Hay que actualizar cada uno de los caminos supervivientes hasta i en $t_{x+1,}$ porque sólo pueden tener L posiciones. De manera que hay que eliminar una, la correspondiente a $t_{x-(L-1)}$, para así poder añadir la correspondiente a t_{x+1}. Para ello se sigue este proceso:

   a) En t_x todos los caminos supervivientes están almacenados en la memoria del decodificador. Tienen L posiciones cada uno, desde la primera $t_{x-(L-1)}$, hasta la última t_x.

   b) Al llegar un nuevo dato al decodificador se añade una nueva etapa al trellis, de manera que la última posición del trellis pasa a ser la t_{x+1}.

   c) El camino superviviente hasta i en t_{x+1} consiste en el camino hasta el estado anterior seleccionado, j ó h en t_x, al que se le añade la rama ji ó hi que hace pasar de t_x a t_{x+1}. En este momento cada camino tiene L +1 posiciones. Desde $t_{x-(L-1)}$ hasta t_{x+1}. Por tanto:

      - Si se ha seleccionado j como estado anterior a i, $DecisionValue_i[t_{x+1}]= j$, entonces el camino superviviente hasta i en t_{x+1} es:
        $Camino_i(0:L)[x+1] = Camino_j(0:(L-1))[x]$ & bit m[n] de la rama ji con origen en t_x y final en t_{x+1}. & representa una concatenación de vectores.

      - Si $DecisionValue_i[x+1]= h$, entonces el camino superviviente hasta i en t_{x+1} es:
        $Camino_i(0:(L))[t_{x+1}] = Camino_h(0:(L-1))[t_x]$ & bit m[n] de la rama hi con origen en t_x y final en t_{x+1}.

5. Los pasos anteriores se repiten para cada uno de los nodos en t_{x+1}. De manera que se obtienen 2^m caminos supervivientes, uno hasta cada estado. Cada uno consta de L+1 posiciones, por eso aún no se pueden almacenar en la FIFO survivor memory.

6. Se selecciona, entre los 2^m caminos supervivientes en t_{x+1}, el superviviente ganador. Es aquel que tenga una menor distancia en su última posición, en este caso t_{x+1}.

   - En primer lugar hay que determinar cuál es la distancia mínima en t_{x+1}:
     $$MinDis\tan ce[t_{x+1}] = \underset{0\le i\le 2^m-1}{\text{Min}}\left(Dis\tan ce_i[t_{x+1}]\right)$$

   - Después se selecciona el índice del camino superviviente ganador i_g:
     $$i_g = \min\left[i \in [0..2^m-1]\right]_{MinDis\tan ce[t_{x+1}]=\underset{0\le i\le 2^m-1}{Min}\left(Dis\tan ce_i[t_{x+1}]\right)}$$
     i_g es el índice del camino que hace que se cumpla la expresión:
     $$MinDis\tan ce[t_{x+1}] = \underset{0\le i\le 2^m-1}{\text{Min}}\left(Dis\tan ce_i[t_{x+1}]\right)$$

   - En el caso de que la distancia mínima no sea única, sino que sea compartida por varios caminos, se elige aleatoriamente uno de ellos como superviviente ganador. En el ejemplo de la *figura 2.15* la distancia mínima es 2, y la comparten los caminos 0, 1 y 2. Por tanto se puede elegir cualquiera de los tres como ganador. En la *figura 2.17*, la distancia mínima es 2, compartida por los caminos 0 y 2. En este caso se puede elegir cualquiera de los dos como ganador.

7. El bit decodificado final DoutV[n=x+1] es el primer bit del camino superviviente ganador en t_{x+1}. Por tanto es igual a:

   - Si el estado seleccionado anterior a i_g en t_{x+1} es el j, entonces: DoutV[n=x+1] = $Camino_{ig}(0)[t_{x+1}]$= $Camino_j(0)[t_x]$.

   - Si el estado seleccionado anterior a i en t_{x+1} fuese el h, entonces: DoutV[n=x+1]= $Camino_{ig}(0)[t_{x+1}]$=$Camino_h(0)[t_x]$.

8. DoutV[x+1] corresponde a la decodificación del dato $DinV_{x-(L-1)}$. Por tanto la latencia del decodificador es L. En la secuencia de salida es el dato $DoutV_{x-(L-1)}$.

9. Una vez obtenido el bit decodificado se guardan los caminos supervivientes en la memoria. Para ello hay que eliminar la primera posición, de manera que se queden con L posiciones:

   - Si el estado seleccionado anterior a i en t_{x+1} es el j, entonces: $Camino_i(0 :(L-1))[t_{x+1}]$ = $Camino_j(1 :(L-1))[t_x]$ & bit m[n] de la rama ji con origen en t_x y final en t_{x+1}.

   - Si el estado seleccionado anterior a i en t_{x+1} es el h, entonces: $Camino_i(0 :(L-1))[t_{x+1}]$ = $Camino_h(1 :(L-1))[t_x]$ & bit m[n] de la rama hi con origen en t_x y final en t_{x+1}.

   - Como el decodificador sólo puede almacenar L posiciones, la etapa t_{x-L} a $t_{x-(L-1)}$ sale de cada uno de los caminos. Entonces en t_{x+1} los caminos supervivientes que se almacenan en la memoria del decodificador, tienen como primera posición la $t_{x-(L-2)}$ y como última la t_{x+1}.

10. Las acciones a realizar entre t_x y t_{x+1} han finalizado, por lo que se vuelve al paso 1.

Nota 2.2: Los caminos en t_{x+1} dependen de los caminos previos en t_x. Entonces el decodificador necesita dos memorias FIFO diferentes. En una de ellas están almacenados todos los valores de $Camino_i[x]$, para todo $i \in [0..2^m-1]$. Estos caminos se utilizan como referencia y el decodificador irá calculando $Camino_i[x+1]$ a partir de ellos. Al finalizar el período de proceso, todos los valores de $Camino_i[x + 1]$, para todo $i \in [0..2^m-1]$, deben estar almacenados en otra FIFO diferente. De esta manera en el siguiente período, entre t_{x+1} y t_{x+2}, una memoria contendrá los caminos de referencia correspondientes a t_{x+1}, y en la otra se irán almacenando los correspondientes a t_{x+2}.

Nota 2.3: En caso de igualdad de distancias se elige aleatoriamente cuál es el estado anterior ganador j ó h, o el camino superviviente ganador. Esta decisión puede ser correcta o errónea, pero no existe ningún método que determine cuál es la mejor decisión. Por este motivo en algunas ocasiones el decodificador tomará la decisión correcta y en otras la errónea. Una consecuencia importante es que la secuencia de salida de dos decodificadores Viterbi puede ser diferente. Ambos implementan el mismo algoritmo pero en caso de igualdad de distancias cada decodificador puede elegir una rama o camino diferentes. Y esa diferencia en la decisión puede ocasionar que la secuencia de salida de ambos sea diferente. Es importante indicar que esto no es un error, sino que es una característica del algoritmo.

2.6.2 Inicio de la trama de entrada empleando intercambio de registros.

La característica fundamental de este estado es que la malla trellis aún no está completa, tiene menos de L etapas. En cada período de proceso llega un dato DinV al decodificador y se introduce en el trellis, añadiendo una etapa más a la malla. Esto se repite hasta t_L, momento en el que la malla está completa y se pasa al caso general. Otra característica importante es que describimos un decodificador cuyo trellis comienza en el estado 0, con distancia 0, en el instante inicial t_0. También recordamos que en esta fase de la decodificación no hay valores en DoutV[t_x]. Aún no ha transcurrido el tiempo de latencia del decodificador, por lo que no hay bits en la salida.

Un período de proceso genérico entre t_x y t_{x+1} consta de dos acciones fundamentales:

A. En t_x la malla trellis está actualizada, su última posición es t_x. Hay que añadir el símbolo que llega a la entrada del decodificador, $DinV_{x+1}$. Para que en el siguiente instante de tiempo, t_{x+1}, la malla esté actualizada y la última posición sea t_{x+1}.

B. En t_x los caminos supervivientes están almacenados en la memoria del decodificador y su última posición corresponde a t_x. Entonces hay que actualizar los caminos para que cuando se produzca el siguiente instante, t_{x+1}, los caminos estén actualizados y su última posición corresponda a t_{x+1}.

Los pasos para realizar las dos acciones anteriores son:

1. Decimos que un estado está inicializado cuando hay al menos una rama o camino que llegue hasta él desde un estado anterior. Si un nodo no está inicializado no pueden partir ramas de él, y tampoco existe distancia hasta ese estado.

2. En t_0 llega $DinV_1$. La malla trellis está vacía y solamente está inicializado el estado 0, con distancia=0 en t_0. De él parten 2^k ramas hasta sus siguientes estados en t_1.

3. Se calcula la distancia hasta llegar a los 2^k estados en t_1, y los caminos supervivientes hasta alcanzar los nodos. Esos 2^k nodos estarán inicializados en t_1, puesto que llega una rama hasta ellos.

4. En t_1 llega $DinV_2$ y de cada uno de los 2^k nodos inicializados parten 2^k ramas hasta sus siguientes estados en t_2. Así que en t_2 habrá 2^{2k} nodos inicializados.

5. Este proceso continúa hasta que los 2^m nodos estén inicializados. Para ello, si k=1, se necesitan m etapas del trellis. Por tanto para $n \geq m$ estarán todos los estados inicializados. En un instante genérico t_x, si $x \leq m$, hay 2^{xk} estados inicializados.

6. Entre t_m y t_{L-1} se continúa en esta fase inicial de la decodificación, porque todos los estados están ya inicializados, pero aún hay menos de L etapas en la malla trellis.

7. El proceso de añadir etapas al trellis, calcular las distancias hasta cada nodo y actualizar los caminos supervivientes, es igual al del modo general. Lo único diferente es que en esta ocasión al añadir nuevas etapas al trellis y bits a los caminos, no hay que eliminar el que ocupaba la primera posición, etapa $t_{x-(L-1)}$ a t_{x-L}.

2.6.3 Final de la trama de entrada empleando truncamiento de trellis.

Abarca el intervalo desde t_N hasta t_{N+L}. En t_{N-1} llega el último símbolo de la trama de entrada al decodificador, y en t_N ese símbolo ya está situado en el trellis. Pero el decodificador continúa trabajando pese a que ya no haya datos en su entrada. Porque aún faltan por decodificar L datos, correspondientes a $DinV_{N-(L-1)}$ a $DinV_N$.

Las características fundamentales de esta fase de la decodificación son:

A. En t_N las distancias hasta cada uno de los estados en la malla trellis están actualizadas. También estarán formados los 2^m caminos supervivientes hasta cada nodo i en t_N. En t_N se obtiene en DoutV[t_N], la decodificación del símbolo correspondiente a $DinV_{(N-L)}$. Todas estas acciones se habrán realizado en el período de proceso de t_{N-1} a t_N, que constituye la última etapa del modo general del decodificador, explicado en *2.6.1*.

B. Como en este modo de funcionamiento no llegan nuevos datos al trellis, no hay que calcular distancias, ni añadir etapas al trellis, ni actualizar los caminos supervivientes. En t_N están almacenados los 2^m caminos supervivientes en la FIFO survivor memory. Y no se modifican nada en el intervalo desde t_N hasta t_{N+L}.

C. Los L símbolos que aún debe obtener el decodificador en DoutV corresponden a los kL bits del camino superviviente ganador en t_N. DoutV[t_{N+1}] corresponde al primer símbolo del camino superviviente ganador en t_N; DoutV[t_{N+2}] corresponde al segundo símbolo del camino. Y así sucesivamente, hasta llegar a DoutV[N+L], que corresponde a la última posición del camino superviviente ganador en t_N.

D. En todo el intervalo, desde t_N hasta t_{N+L}, no se modifican los caminos ni las distancias. Por tanto en t_N ya se dispone de los L símbolos que se obtendrán en DoutV[n]. Entonces sería posible disponer de todos ellos en la salida, separados por un T_{CLK}, sin embargo esto modificaría el throughput del sistema y sería perjudicial. Por este motivo el decodificador almacena los datos en su memoria y los va enviando a la salida con la misma frecuencia que en el modo general, un símbolo en la salida por cada período de proceso.

Los pasos a seguir son:

1. Se selecciona el camino superviviente ganador en t_N.
   - $MinDis\tan ce[t_N] = \underset{0\le i\le 2^m-1}{\text{Min}}\left(Dis\tan ce_i[t_N]\right)$
   - El índice del camino superviviente ganador es:
     $i_{gN} = \min\left[i \in [0..2^m-1]\right]_{MinDis\tan ce[t_N]=\underset{0\le i\le 2^m-1}{Min}(Dis\tan ce_i[t_N])}$
   - i_{gN} es el índice que cumple: $MinDis\tan ce[t_{x+1}] = \underset{0\le i\le 2^m-1}{\text{Min}}\left(Dis\tan ce_i[t_{x+1}]\right)$

2. En t_{N+1} se obtiene DoutV[N+1], que es el primer bit del camino ganador en t_N. DoutV[n=N+1]=$camino_{igN}$(0)[n=N];

3. En t_{N+2} se obtiene DoutV[N+2], que es el segundo bit del camino ganador en t_N.
   DoutV[n=N+2]=camino$_{igN}$(1)[n=N];

4. El proceso continúa, hasta que en t_{N+L}, se obtiene el último bit decodificado. Que corresponde al último bit del camino ganador en t_N.
   DoutV[n=N+L]=camino$_{igN}$(L-1)[n=N];

5. La decodificación de la secuencia de entrada $DinV_1$...$DinV_N$ ha finalizado.

2.6.4 Método traceback. Comparativa con intercambio de registros.

Se trata de los dos métodos más utilizados. Tienen sus ventajas e inconvenientes, que son las siguientes:

La mayor ventaja del intercambio de registros es que se consigue una frecuencia de proceso de un dato mayor que con el traceback. Además la latencia también es menor. Pero el inconveniente es que en cada período de proceso hay que actualizar por completo todos los caminos supervivientes, obteniendo otros 2^m caminos completamente diferentes. Entonces hay que reescribir todas las posiciones de los 2^m caminos en cada período de proceso y esto requiere muchos accesos a memoria. La consecuencia negativa de esto es que se requiere un área y un consumo de potencia mayor que en el método traceback. El área será mayor que con el método traceback, pero se mantiene en niveles razonables si no se utilizan valores altos de constraint length, K<9. Si se utiliza K alto, K≥9, entonces el área crece mucho y este sistema no se suele emplear porque resulta muy costoso en términos de área.

Por contra el método traceback tiene como principal ventaja que ocupa menos área y consumo. Pero los inconvenientes son que la frecuencia de proceso de un dato es menor y la latencia es mayor. Otro inconveniente añadido es que la frecuencia de proceso de un dato depende de L, cuanto mayor sea L, menor será la frecuencia. En cambio con intercambio de registros la frecuencia es independiente de L. Por todos estos motivos, este método suele emplearse en sistemas donde la velocidad de decodificación no es un factor fundamental.

En nuestro proyecto utilizamos intercambio de registros porque la K es sólo 7. Así conseguimos mayor velocidad, independencia con L y todo ello dentro de un consumo de área y potencia razonables, porque tenemos K<9. Además lo hemos diseñado optimizando tanto el área como la velocidad y hemos cumplido con los objetivos del proyecto, que detallamos en el *tema 7*.

A continuación explicamos brevemente la técnica traceback. No nos extendemos porque no es de interés para el proyecto, y porque puede ampliarse información en la bibliografía citada en *2.8.2D* y *2.8.5*.

En cada etapa del trellis hay que almacenar el estado, j ó h ganador, anterior a cada estado i. Y también el bit m[n] de la rama ganadora que transiciona desde j ó h en t_x hasta i en t_{x+1}. La ventaja de este sistema es que al añadir una nueva etapa al trellis no hay que modificar todas las posiciones de memoria. Es suficiente con añadir la nueva etapa, que abarca de t_x a t_{x+1}, y sacar la primera posición de la memoria, que abarca de

t_{x-L} a $t_{x-(L-1)}$. Por tanto es una estructura FIFO. El proceso de actualización se ve en las siguientes tablas, en la que están actualizados los caminos del ejemplo desarrollado en el *apartado 2.7*, correspondiente al período de proceso entre n=6 y n=7.

La FIFO survivor memory consta de m filas. Cada fila tiene L posiciones. Y en cada posición se almacena el estado j ó h ganador, anterior a i, y el bit m[n] de la rama ganadora ji ó hi que transiciona hasta el estado i en t_{x+1}.

Tabla 2.10: Contenido de la memoria con método traceback, en t_6.						
	Tiempo t_x ó n=x	2	3	4	5	6
	Posición Memoria	0	1	2	3	L-1
Estado i		Estado anterior ganador j ó h/bit m[n] de la rama ganadora ji ó hi.				
S_0=00		00/0	00/0	01/0	01/0	00/0
S_1=01		10/0	11/0	11/0	10/0	10/0
S_2=10		00/1	01/1	01/1	00/1	00/1
S_3=11		**10**/**1**	11/1	10/1	10/1	11/1

Tabla 2.11: Contenido de la memoria con método traceback, en t_7.					
Tiempo t_x ó n=x	3	4	5	6	7
Posición Memoria	0	1	2	3	L-1
Estado i	Estado anterior ganador j ó h/bit m[n] de la rama ganadora ji ó hi.				
S_0=00	00/0	**01**/0	01/0	00/0	00/0
S_1=01	**11**/0	11/0	10/0	**10**/0	10/0
S_2=10	01/1	01/1	**00**/1	00/1	**01**/1
S_3=11	11/1	10/1	10/1	11/1	10/1

Para obtener el símbolo decodificado DoutV[n=7], hay que realizar L+1 lecturas de memoria, comenzando en t_7 y finalizando en t_2, el proceso es el siguiente:

- En t_7 el estado ganador es el 10, con distancia 2. Entonces 10 es el índice para la primera lectura, en t_7.
- Lectura 1: El estado anterior al 10 en t_7 es el 01 en t_6.
- Lectura 2: El estado anterior al 01 en t_6 es el 10 en t5.
- Lectura 3: El estado anterior al 10 en t_5 es el 00 en t_4.
- Lectura 4: El estado anterior al 00 en t_4 es el 01 en t_3.
- Lectura L: El estado anterior al 01 en t_3 es el 11 en t_2.
- Lectura L+1: Se obtiene DoutV[n=7] =´1´. Se obtiene leyendo el bit m[n] de la posición correspondiente a t_2. *Nota 2.4*: En el instante t_7, la posición t_2 no está en la memoria, porque se elimina en el período de proceso que transcurre entre t_6 y t_7. Por ello en el período de proceso se deben realizar las lecturas antes de eliminar la primera posición de la FIFO.

Así queda clara la desventaja de este método, se necesitan L+1 lecturas de memoria en cada período de proceso. Y deben realizarse en serie, porque en cada lectura se obtiene el índice de la siguiente. Con el intercambio de registros se necesitan 2^m escrituras en cada período de proceso, pero pueden realizarse en paralelo. Por eso se consigue más velocidad a costa de aumentar el área.

2.7 Ejemplo práctico de codificación-decodificación con 4 estados.

A continuación desarrollamos un ejemplo completo de codificación, transmisión, recepción y decodificación de una secuencia. Aplicaremos ruido a la secuencia y observaremos como el decodificador consigue corregir todos los errores causados por el ruido. Para el ejemplo utilizamos el codificador convolucional (2, 1,K=3). Notas:

- El codificador tiene retardo cero.
- En el instante inicial el codificador parte del estado 00. Así que es necesario resetear al codificador antes de que llegue el primer bit de una trama de entrada.
- Utilizamos método intercambio de registros, truncamiento de trellis y el decodificador comienza en el estado 0 con distancia 0 en t_0.

La secuencia con información fuente es m[n]={1,1,0,0,1,0,1,0}={m[0],m[1],.....m[6],m[7]}. Por tanto contiene 8 símbolos, N=8; m[n]={m_1, m_2....m_N}.

Tabla 2.12: Codificación secuencia ejemplo. Codificador (2, 1,K=3).								
Instante	t_0, n=0	t_1, n=1	t_2, n=2	t_3, n=3	t_4, n=4	t_5, n=5	t_6, n=6	t_7, n=7; t_{N-1}, n=N-1
m[n]	1	1	0	0	1	0	1	0
m[n-1]m[n-2]	00	10	11	01	00	10	01	10
c_1[n] c_0[n]	11	10	10	11	11	01	00	01

Entonces, la secuencia codificada es c[n]=(c_1[n],c_0[n])={11,10,10,11,11,01,00,01} ={c[0],c[1],c[2],c[3], c[4],c[5],c[6],c[N-1]}.

c[n] se transmite y se obtiene: DinV[n]=(r_1[n],r_0[n])={11,10,00,10,11,01,00,01} ={DinV[0],DinV[1], ..., DinV[N-1]}={$DinV_1$,....,$DinV_N$}.

Debido al ruido del canal la secuencia recibida no es igual a la transmitida retardada: DinV[n] ≠ c[n- $retardo_{canal}$]. Concretamente se han producido dos errores:

1. c[2]= 10 ≠ DinV[2]= 00
2. c[3]= 11 ≠ DinV[3]= 10

Pero al desarrollar el ejemplo veremos que en la salida del decodificador se obtiene DoutV[n]={1,1,0,0,1,0,1,0}= {DoutV[0], DoutV[1], ..., DoutV[N-1=7]}={$DoutV_1$, $DoutV_2$, ..., $DoutV_{N=8}$}=m[n- ($retardo_{canal}$ +latencia)]. Así que el decodificador ha cumplido el objetivo y ha corregido los errores en la transmisión.

En la *tabla 2.13* se muestran todas las secuencias implicadas en el ejemplo. Siendo exactos, el primer dato llega al decodificador en el instante n = $retardo_{canal}$, y el primer dato en la salida se obtiene en n = ($retardo_{canal}$ + latencia). Pero por simplicidad consideramos que en todas las señales implicadas el primer símbolo llega en n=0, t_0, el segundo en n =1, t_1 y así sucesivamente.

Tabla 2.13: Codificación secuencia ejemplo. Codificador (2, 1,K=3).								
Instante	t_0, n=0	t_1, n=1	t_2, n=2	t_3, n=3	t_4, n=4	t_5, n=5	t_6, n=6	t_7, n=7; t_{N-1}, n=N-1
m[n]	1	1	0	0	1	0	1	0
C[n]	11	10	10	11	11	01	00	01
DinV[n]	11	10	00	10	11	01	00	01
DoutV[n]	1	1	0	0	1	0	1	0

2.7.1 Inicio de la trama de entrada.

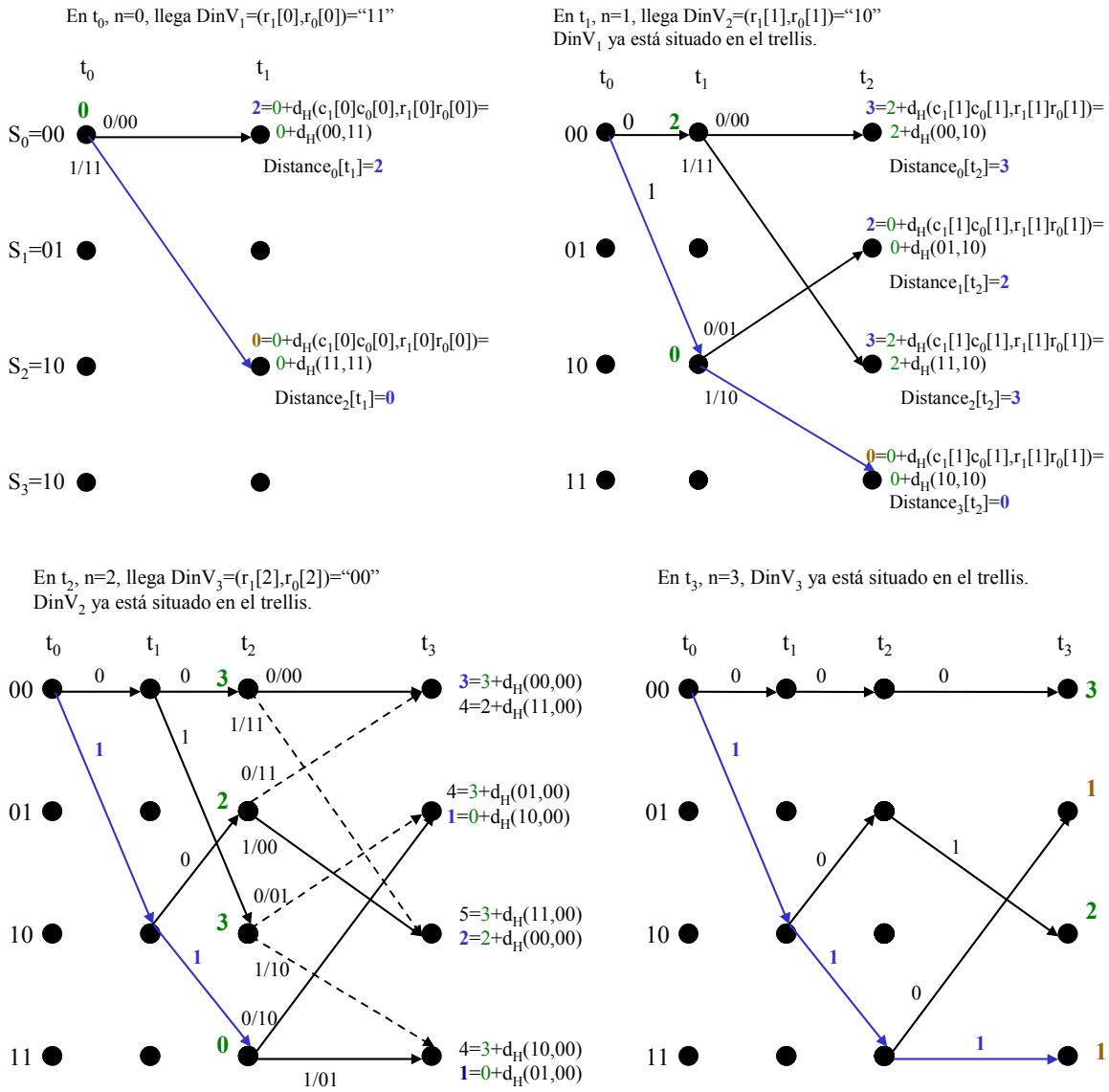


Figura 2.18: Ejemplo de decodificación entre n=0 y n=3.

Tabla 2.14: Decodificación en el intervalo [n=0...n=3].				
	t_0, n=0	t_1, n=1	t_2, n=2	t_3, n=3
$DinV[n]=r_1[n]r_0[n]$	$DinV_1$=11	$DinV_2$=10	$DinV_3$=00	$DinV_4$=10
$Camino_0(0:4)[n]$	Vacío	bit(0)=0 bits(1:4)--> Vacío	bits(0:1)=00 bits (2:4)--> Vacío	bits(0:2)=000 bits (3:4)--> Vacío
$Camino_1(0:4)[n]$	Vacío	bits(0:4)--> Vacío	bits (0:1)=10 bits (2:4)--> Vacío	bits (0:2)=110 bits(3:4)--> Vacío
$Camino_2(0:4)[n]$	Vacío	bit(0)=1 bits(1:4)--> Vacío	bits (0:1)=01 bits (2:4)--> Vacío	bits(0:2)=101 bits (3:4)--> Vacío
$Camino_3(0:4)[n]$	Vacío	bits(0:4)--> Vacío	bits (0:1)=11 bits(2:4)--> Vacío	bits(0:2)=111 bits(3:4)--> Vacío
DoutV[n]	No hay datos	No hay datos	No hay datos	No hay datos
$Distance_0[n]$	0	2	3	3
$Distance_1[n]$	No hay datos	No hay datos	2	1
$Distance_2[n]$	No hay datos	0	3	2
$Distance_3[n]$	No hay datos	No hay datos	0	1

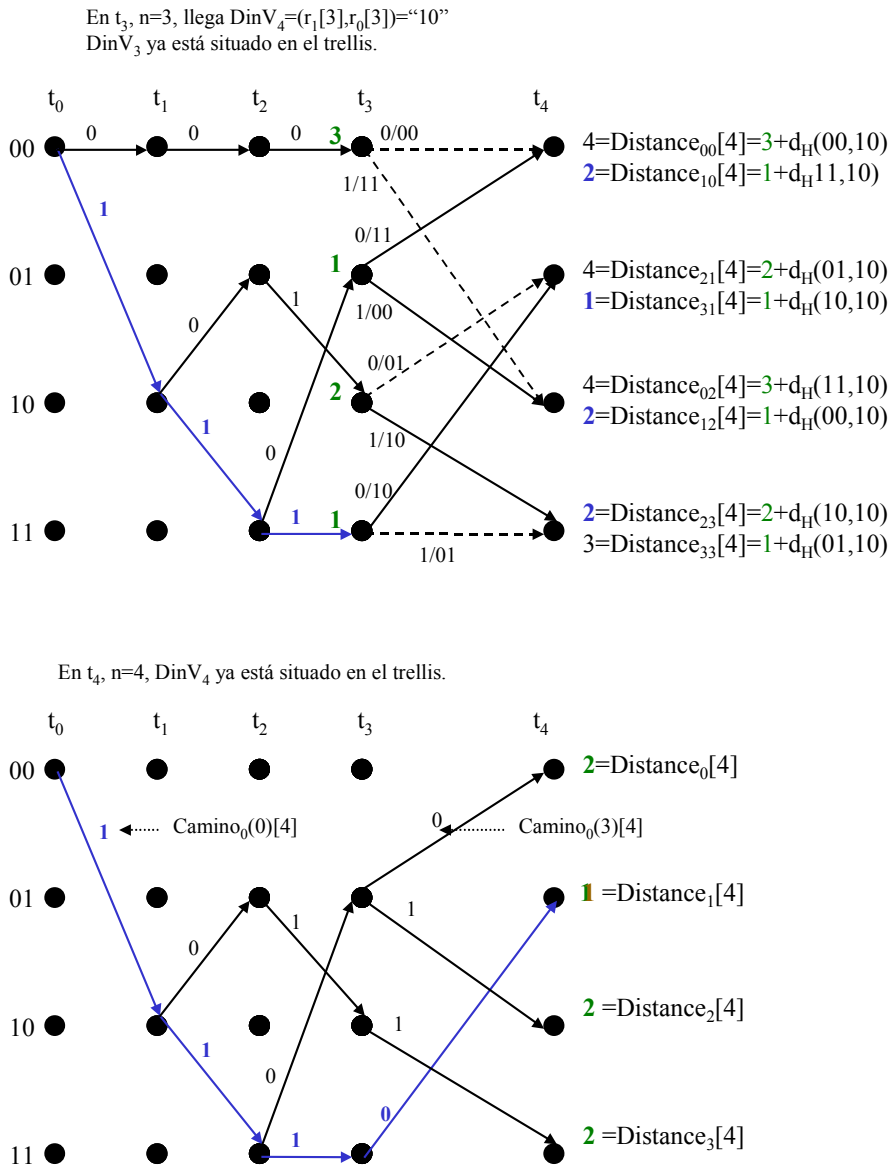


Figura 2.19: Ejemplo de decodificación entre n=3 y n=4.

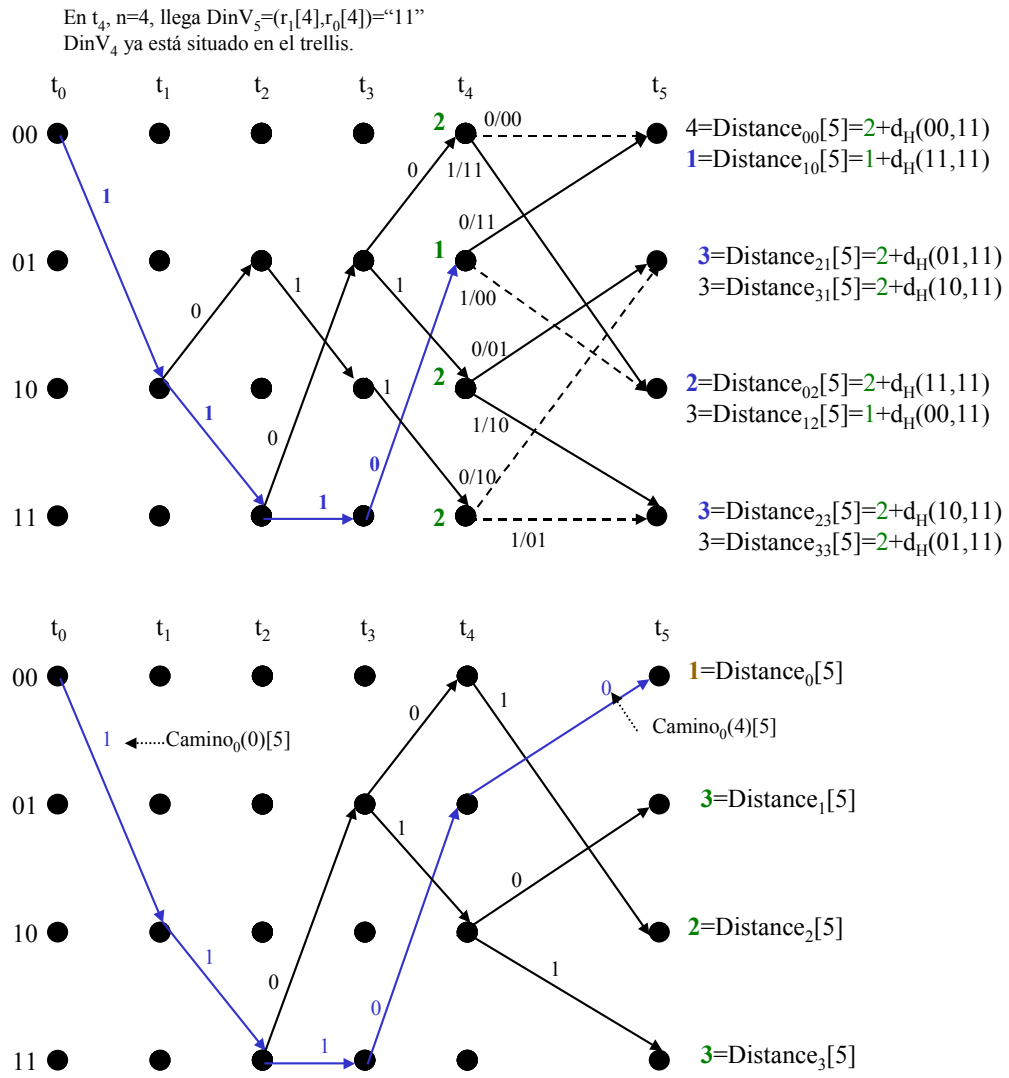


Figura 2.20: Ejemplo de decodificación entre n=4 y n=5.

Tabla 2.15: Decodificación en el intervalo [n=3...n=5].			
	t_3, n=3	t_4, n=4	t_5, n=5
$DinV[n]=r_1[n]r_0[n]$	$DinV_4$=10	$DinV_5$=11	$DinV_6$=01
Camino$_0$(0:4)[n]	bits(0:2)=000 bits (3:4)--> Vacío	bits(0:3)=1100 bit(4)-->Vacío	**11100** superviviente ganador
Camino$_1$(0:4)[n]	bits (0:2)=110 bits(3:4)--> Vacío	bits(0:3)=**1110** bit(4)-->Vacío	11010
Camino$_2$(0:4)[n]	bits(0:2)=101 bits (3:4)--> Vacío	bits(0:3)=1101 bit(4)-->Vacío	11001
Camino$_3$(0:4)[n]	bits(0:2)=**111** bits(3:4)--> Vacío	bits(0:3)=1011 bit(4)-->Vacío	11011
DoutV[n]	No hay datos	No hay datos	No hay datos
Distance$_0$[n]	3	2	**1**
Distance$_1$[n]	**1**	**1**	3
Distance$_2$[n]	2	2	2
Distance$_3$[n]	**1**	2	3

2.7.2 Caso general.

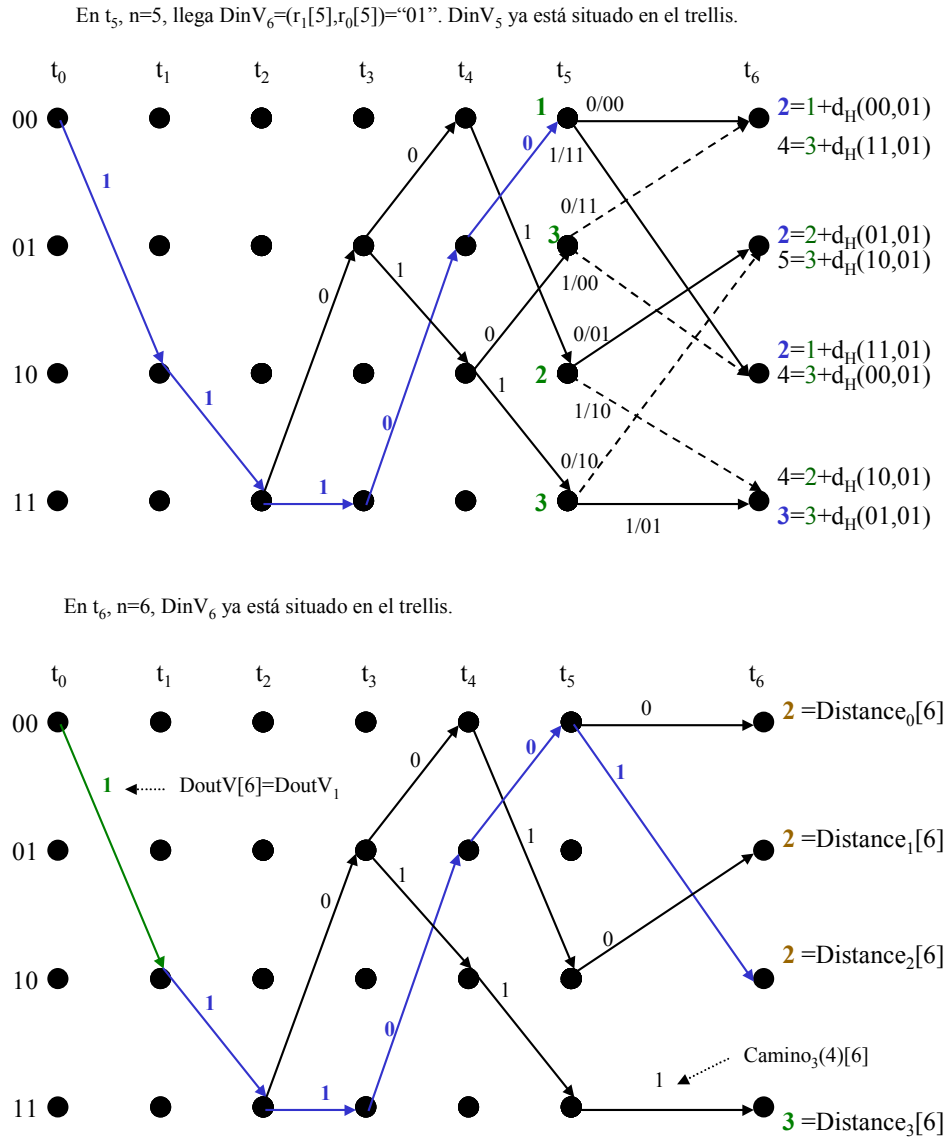


Figura 2.21: Ejemplo de decodificación entre n=5 y n=6.

En n=6 se obtiene el primer bit de la secuencia de salida del decodificador: $DoutV_1$=DoutV[6]='1'. Este bit corresponde a la decodificación del primer símbolo de entrada al decodificador, $DinV_1$.

En este ejemplo en concreto se ha elegido como camino superviviente ganador el 2. Pero podrían haberse elegido también el 0 ó el 1. La decisión es aleatoria, pero en este caso no afecta al resultado de la decodificación, porque el inicio de todos los caminos converge en una misma rama. *Nota 2.5*: En este caso concreto convergen, pero no se trata de una condición obligatoria. Puede darse la situación de que los caminos no converjan y al decidir aleatoriamente se tome una decisión incorrecta.

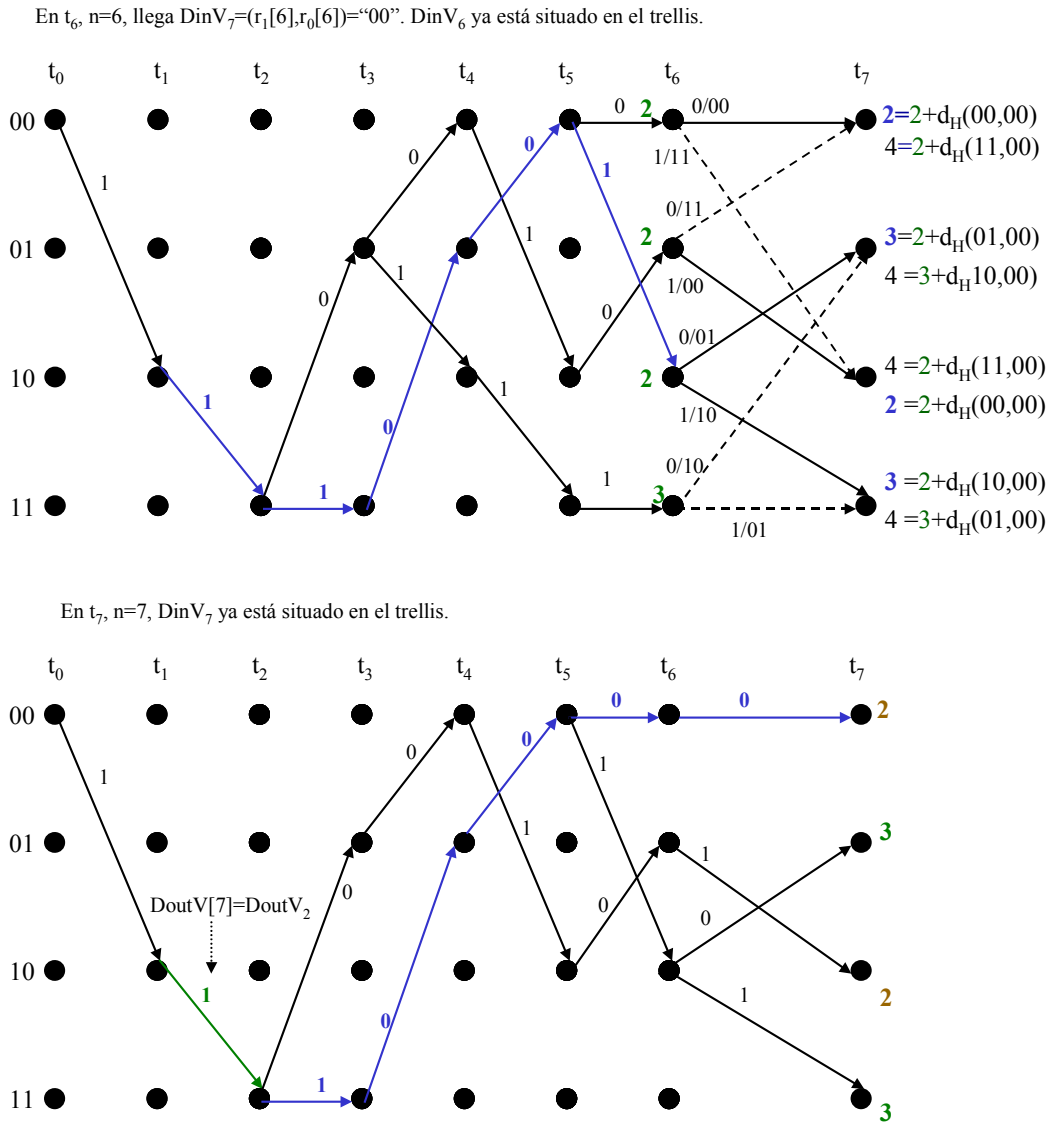


Figura 2.22: Ejemplo de decodificación entre n=6 y n=7.

En n=7 se obtiene el segundo bit de la secuencia de salida del decodificador: $DoutV_2$=DoutV[7]='1'. Este bit corresponde a la decodificación del segundo símbolo de entrada al decodificador, $DinV_2$.

Se puede elegir como camino superviviente ganador el cero o el dos indistintamente. En la *figura 2.22* se ha elegido el cero, pero en la *2.17* se eligió el 2. Ambas opciones son correctas.

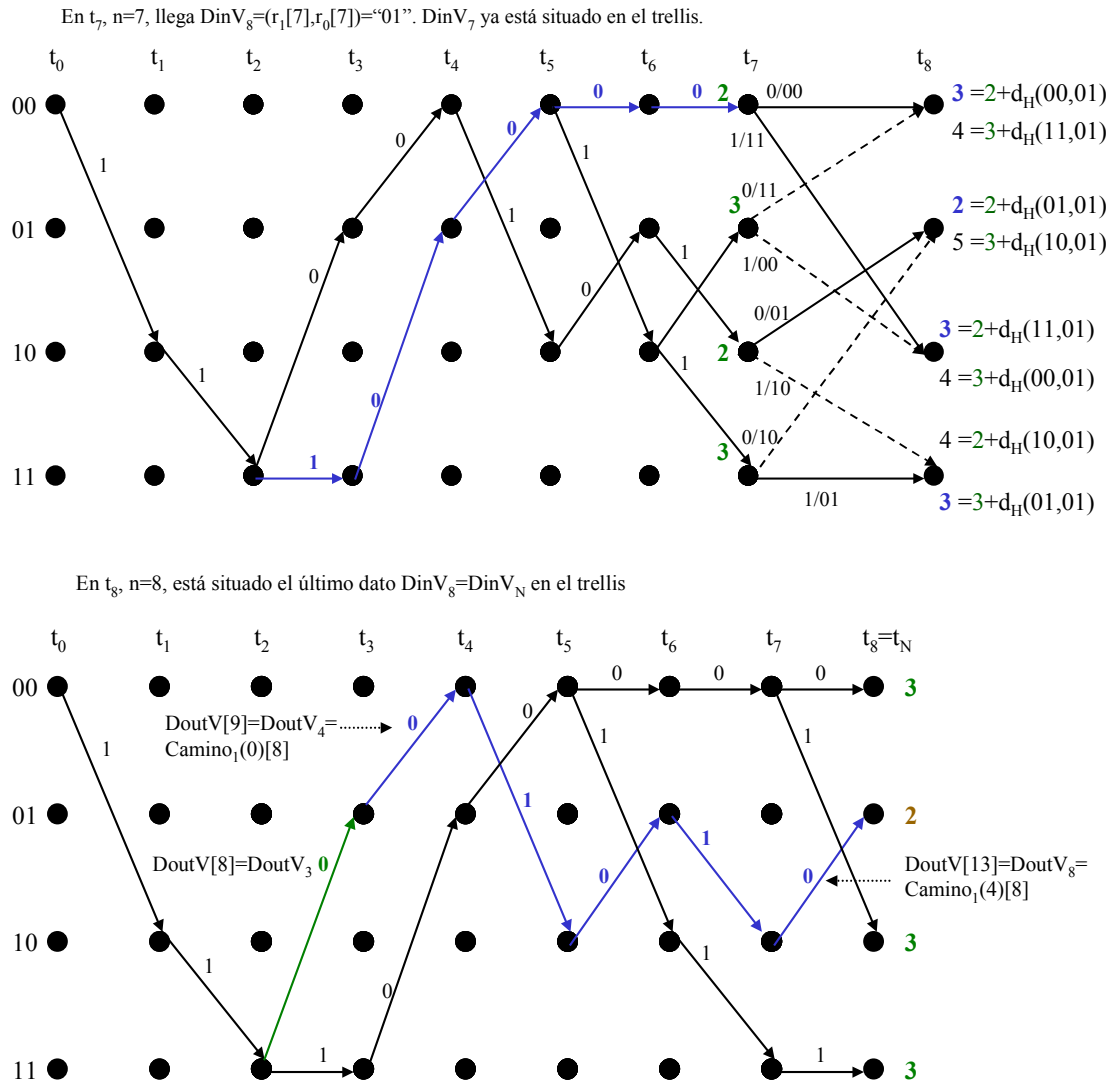


Figura 2.23: Ejemplo de decodificación entre n=7 y n=8.

Tabla 2.16: Decodificación en el intervalo [n=5...n=8].				
	t_5, n=5	t_6, n=6	t_7, n=7; t_{N-1}, n=N-1	t_8, n=8; t_N, n=N
$DinV[n]=r_1[n]r_0[n]$	$DinV_6$=01	$DinV_7$=00	$DinV_{N=8}$=01	No hay datos Ha finalizado la secuencia de entrada
$Camino_0(0:4)[n]$	**11100**	11000	**10000**	00000
$Camino_1(0:4)[n]$	11010	10010	10010	**01010**
$Camino_2(0:4)[n]$	11001	**11001**	00101	00001
$Camino_3(0:4)[n]$	11011	10111	10011	00111
DoutV[n]	No hay datos	$DoutV_1$=**1** $Camino_0(0)[5]$	$DoutV_2$=**1** $Camino_0(0)[6]$	$DoutV_3$=**0** $Camino_2(0)[7]$
$Distance_0[n]$	**1**	**2**	**2**	3
$Distance_1[n]$	3	**2**	3	**2**
$Distance_2[n]$	2	**2**	**2**	3
$Distance_3[n]$	3	3	3	3

2.7.3 Final de la trama de entrada. Resultados.

En n=N-1=7 llega el último dato $DinV_{N=8}$ a la entrada del decodificador. En n=N el símbolo está situado en el trellis. A partir de este momento ya no se modifica la malla trellis, ni hay que calcular distancias ni modificar los caminos. El trabajo del decodificador consiste en sacar uno a uno los bits del camino superviviente ganador en n=N.

Tabla 2.17: Decodificación del final de la trama.

	t_8, n=8; t_N	n=9;	n=10	n=11	n=12	n=13; n=N+L
DoutV[n]	$DoutV_3$=**0** $Camino_2(0)$[7]	$DoutV_4$=**0** $Camino_1(0)$[N=8]	$DoutV_5$=**1** $Camino_1(1)$[N]	$DoutV_6$=**0** $Camino_1(2)$[N]	$DoutV_7$=**1** $Camino_1(3)$[N]	$DoutV_{N=8}$=**0** $Camino_1(4)$[N]
$DinV[n]=r_1[n]r_0[n]$	No hay datos. Ha finalizado la secuencia de entrada					
$Camino_0(0:4)[n]$	00000, constante para n ≥ N.					
$Camino_1(0:4)[n]$	**01010**, constante para n ≥ N.					
$Camino_2(0:4)[n]$	00001, constante para n ≥ N.					
$Camino_3(0:4)[n]$	00111, constante para n ≥ N.					
$Distance_0[n]$	3, constante para n ≥ N.					
$Distance_1[n]$	2, constante para n ≥ N.					
$Distance_2[n]$	3, constante para n ≥ N.					
$Distance_3[n]$	3, constante para n ≥ N.					

La secuencia final DoutV[n]={1,1,0,01,0,1,0}={$DoutV_1$, $DoutV_2$, … , $DoutV_{N=8}$,} es exactamente igual a la original m[n]. Por tanto en la salida del decodificador la BER es cero, y el sistema FEC ha conseguido el objetivo de disminuir la BER.

$$BERoutDecoder = \frac{Bitsdiferentes\left(m\left[n-(retardo_{canal}+Latencia)\right], DoutV[n]\right)}{NumBits\left(m[n]\right)} = 0$$

Sin embargo en la entrada del decodificador, la BER no es cero.

$$BERinDecoder = \frac{\text{BitsDiferentes}\left(c_1\left[n-retardo_{canal}\right], r_1[n]\right)+\text{BitsDiferentes}\left(c_0\left[n-retardo_{canal}\right], r_0[n]\right)}{2*\text{Número de bits}\left(c_1[n]\right)} =$$

$$\frac{2}{16} = 0.125$$

2.8 Referencias.

Proporcionamos el enlace Web siempre que exista, accedemos a estos enlaces por última vez en noviembre de 2011. Además tenemos una copia de seguridad en el CD del proyecto de toda la documentación sin copyright.

Dividimos la bibliografía en varios puntos:

- En el *2.8.1* está la bibliografía que trata de una manera genérica sobre la codificación-decodificación convolucional y el algoritmo de Viterbi. Esta bibliografía es fundamental porque describe de manera exhaustiva todos los aspectos de interés necesarios para nuestro proyecto.

- En el resto de apartados seleccionamos documentos que describen de manera precisa algún tema específico de interés sobre nuestro proyecto. En muchas ocasiones se trata de documentos genéricos, que no sólo describen el tema que hemos seleccionado, sino que también detallan el resto de aspectos de la codificación-decodificación convolucional y del algoritmo Viterbi. Realizamos esta clasificación para facilitar la búsqueda de información al lector, porque la bibliografía es muy extensa. Por eso hemos seleccionando los documentos que a nuestro juicio explican mejor cada tema en concreto.

2.8.1 Genéricas sobre codificación convolucional-decodificación Viterbi.

Todos los libros de este apartado de la bibliografía describen la codificación-decodificación convolucional y el algoritmo Viterbi. Sobre cada uno de ellos indicamos los capítulos y las páginas que tratan sobre este amplio tema. Los conocimientos que proporciona esta bibliografía son fundamentales para poder realizar el proyecto.

Como incluimos mucha bibliografía seleccionamos algunos temas que son particularmente importantes en nuestro proyecto:

A. Codificación convolucional.
B. Ejemplo del proceso de decodificación Viterbi con 4 estados.
C. Estructura módulo ACS.
D. Estructura módulos intercambio de registros y traceback.
E. Calculo teórico de los límites de la probabilidad de error.
F. Decodificación soft.

Cualquiera de las referencias de este apartado describe los 6 temas anteriores. Pero para facilitar el acceso a estos temas, en algunas referencias indicamos las páginas que los describen de manera más precisa. Esto debe interpretarse únicamente como una ayuda para facilitar el acceso a la información, porque citamos muchas referencias y cada una de ellas abarca muchas páginas. Es una manera de clasificar el contenido genérico de la bibliografía según se adapte a nuestras necesidades específicas para el proyecto. Pero es una clasificación flexible, por ejemplo, no significa que la referencia [1] sólo trate sobre el módulo ACS y que la [3] no describa el módulo ACS. Todas las referencias describen los 6 temas y pueden consultarse indistintamente.

[1]Todd K.Moon. "Error Correction Coding. Mathematical Methods and Algorithms".Wiley-Interscience. Capítulo 12, páginas 452-534, publicado en 2005.
[1B] Ejemplo de decodificación Viterbi con 4 estados, páginas 471-481.
[1C] Estructura ACS, página 481.
[1D] Estructura intercambio de registros y traceback, páginas 481-484.
[1E] Calculo teórico de los límites de la probabilidad de error, páginas 491-505.

[2] Bernard Skalar. "Digital Communications, Fundamentals and Applications". Prentice Hall PTR . Capítulo 7, páginas 381-435, segunda edición 21 Enero 2001.
[2A] Codificación convolucional, páginas 382-394.
[2B] Ejemplo de decodificación Viterbi con 4 estados: 401-408.
[2C] Estructura ACS, página 405-408.

[3]Shu Lin and Daniel J. Costello, Jr. "Error Control Coding. Fundamentals and Applications." Prentice-Hall Series in Computer Applications in Electrical Engineering. Capítulos 10 y 11, páginas 343-345, publicado en 1983.
[3A] Codificación convolucional, páginas 295-308.

[4]Andrew J.Viterbi and Jim K. Omura. "Principles of Digital Communication and Coding". McGraw-Hill Series in Electrical Engineering. Capítulos 4 y 5, páginas 227-381, publicado en 1979.
[4A] Codificación convolucional, páginas 227-235.
[4E] Calculo teórico de los límites de la probabilidad de error: 239-348.

[5]Robert H. Morelos-Zaragoza. "The Art of error Correcting Coding". Wiley, capítulos 5, 6 y 7. Páginas 87-168, segunda edición 2006.
[5B] Ejemplo de decodificación Viterbi con 4 estados, páginas 102-108.
[5C] Estructura ACS, páginas: 103, 111 y 112.
[5D] Estructura intercambio de registros y traceback: páginas 109-110.

[6]John G. Proakis. "Digital Communications". McGraw Hill, capítulo 8, páginas 470-533, cuarta edición 2001.
[6A] Codificación convolucional, páginas 470-483.
[6E] Calculo teórico de los límites de la probabilidad de error: 486-500; 506-526.

[7] Chip Fleming. "A Tutorial on Convolutional Coding with Viterbi Decoding". Spectrum Applications 11 Febrero de 2006. cfleming@ieee.org
http://pw1.netcom.com/~chip.f/viterbi/tutorial.html
[7B] Ejemplo de decodificación Viterbi con 4 estados:
http://home.netcom.com/~chip.f/viterbi/algrthms.html
[7D] Estructura intercambio de registros y traceback:
http://home.netcom.com/~chip.f/viterbi/algrthms2.html

[8]Jorge Castiñeira Moreira and Patrick Guy Farrell. "Essentials of Error-Control Coding". Wiley, capítulo 6, páginas 157-208, publicado en 2006.
[8A] Codificación convolucional, páginas 157-181.
[8B] Ejemplo de decodificación Viterbi con 4 estados: 182-185.
[8E] Calculo teórico de los límites de la probabilidad de error: 186-196.
[8F] Decodificación soft: 196-200.

[9] Andrew J. Viterbi. "CDMA Principles of Spread Spectrum Communication". Addison-Wesley Wireless Communications Series. Capítulo 5, páginas 123-177. Publicado en 1995.
[9A] Codificación convolucional, páginas 127-132.
[9C] Estructura ACS, páginas: 155-159.
[9E] Calculo teórico de los límites de la probabilidad de error: 140-155; 159-177.

[10] W. Cary Huffman and Vera Pless. "Fundamentals of error Correcting Codes". Cambridge University Press, capítulos 14 y 15, páginas 546-614, 1 Febrero 2003.
[10A] Codificación convolucional, páginas 546-555.
[10F] Decodificación soft: 573-593.

[11] G. David Forney, JR. "The Viterbi Algorithm". Proceedings of the IEEE, Vol 61, N º 3, March 1973.
http://gladstone.systems.caltech.edu/EE/Courses/EE127/EE127A/handout/ForneyViterbi.pdf

http://www.mendeley.com/research/hidden-markov-models-3-implementing/

[12] Peter Sweeny. "Error control Coding. From Theory to Practice". Wiley, capítulo 2, páginas 35-66. Publicado en 2002.
[12A] Codificación convolucional, páginas 35-41.
[12D] Estructura intercambio de registros y traceback: 45-51.

[13] Alain Glavieux. "Channel Coding in Communication Networks. From Theory to Turbocodes". ISTE Ltd, capítulo 3, páginas 129-196, publicado en 2007.
[13A] Codificación convolucional, páginas 129-140.
[13E] Calculo teórico de los límites de la probabilidad de error: 142-149; 172-196

[14] Khmaies Ouahada. "Viterbi Decoding of Ternary Line Codes". Publicado por VDM Verlag Dr. Müller en 2008. Capítulos 4, páginas 30-37; y capítulos 6, 7, 8 y 9 páginas 56-108. *Nota 2.6*: En este libro se habla principalmente de códigos de tres niveles, cuando nosotros siempre usamos un binario.
[14B] Ejemplo de decodificación Viterbi con 4 estados: 30-37.
[14F] Decodificación soft: 56-80.

[15] G. Kabatiansky; E. Krouk and S. Semenov. "Error Correcting Coding and Security for Data Networks. Analysis of the Superchannel Concept". John Wiley & Sons, Ltd. Capítulo 6, páginas 141-190, publicado en 2005.
[15A] Codificación convolucional, páginas 141-150.
[15B] Ejemplo de decodificación Viterbi con 4 estados: 150-156.
[15F] Decodificación soft: 156-160; 184-189.

[16] Christian B. Schlegel and Lance C. Pérez. "Trellis and Turbo Coding". IEEE Press Series on Digital & Mobile Communication. Capítulo 4, páginas 95-123; capítulos 6 y 7, páginas 155-225. Publicado en 2004.
[16F] Calculo teórico de los límites de la probabilidad de error: 155-182.

2.8.2 Ejemplo 4 estados.

[17] "Example: Viterbi algorithm". Application report: "Viterbi Decoding Techniques in the TMS320C54x Family". Noviembre 2010.
http://es.scribd.com/doc/43577105/Viterbi-Example

[18] Siriram Swaminathan and Russell Teisser. "An FPGA Based Adaptive Viterbi Decoder". Department of ECE University of Massachusetts Amherst. 15 Noviembre 2006.
http://www.ecs.umass.edu/ece/tessier/courses/636/lect21-ece636.pdf

[19] Sh. Sanjay Sharma and Pushpinder Kaur. "Implementation of Low Power Viterbi Decoder on FPGA". Páginas 19-25, Junio 2006.
http://dspace.thapar.edu:8080/dspace/bitstream/123456789/292/1/r8044117.pdf

2.8.3 Arquitectura completa decodificador Viterbi.

[20] Wei Chen. "RTL implementation of Viterbi decoder". Master Tesis desarrollada en Sistemas Electrónicos, departamento de Ingeniería de Computación en la Universidad de LinKöpings. 2 Junio 2006.
http://liu.diva-portal.org/smash/get/diva2:22064/FULLTEXT01

[21] Mohammed Benaissa and Yiqun zhu."A Novel High-Speed Configurable Viterbi Decoder for Broadband Access". EURASIP Journal on Applied Signal Processing,volumen 13, pp. 1317–1327, 2003 Hindawi Publishing Corporation.
http://www.hindawi.com/journals/asp/2003/865460/abs/

http://downloads.hindawi.com/journals/asp/2003/865460.pdf

[22] Siriram Swaminathan; Russell Teisser; Dennis Goeckel and Wayne Burleson. "A Dynamically Reconfigurable Adaptive Viterbi Decoder". Department of Electrical and Computer Engineering. University of Massachusetts. Amherst, MA. 01003. 24- Febrero 2002.
http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.63.6428

[23] Sh. Sanjay Sharma and Pushpinder Kaur. "Implementation of Low Power Viterbi Decoder on FPGA". Páginas 26-29; 34-42; 53-69, Junio 2006.
http://dspace.thapar.edu:8080/dspace/bitstream/123456789/292/1/r8044117.pdf

[24] "Convolutional Codes & Viterbi Algorithm". Bit Engineering Lab., SITI (System Integration Technology Institute). Information and Communications University. Diapositivas 10-38.

[25] Henry Hendrix."Viterbi Decoding Techniques for the TMS320C54x DSP Generation". Texas Instruments, Application Report SPRA071A –January 2002. Páginas 6-20.
http://www.ti.com/lit/an/spra071a/spra071a.pdf

[26] Engling Yeo; Stephanie Augsburger; Wm. Rhett Davis and Borivoje Nikolic. "Implementation of High throughput soft output Viterbi Decoders". Proc. IEEE Workshop on Signal Processing Systems, 146-151, October, 2002.
http://bwrc.eecs.berkeley.edu/php/pubs/pubs.php/325/yeo.pdf

2.8.4 ACS

[27] Herbert Dawid, Olaf J. Joeressen and Heinrich Meyr. "Chapter 17 Viterbi Decoders: High Performance Algorithms and Architectures". Páginas 3-24.
http://www.eecs.berkeley.edu/newton/Classes/EE290sp99/lectures/ee290aSp996_1/vit_chap17.pdf

[28] Man Guo Ahmad; M.Omair Swamy and Chunyan Wang "FPGA design and implementation of a low-power systolic array-based adaptive Viterbi decoder". Publicado en: Circuits and Systems I: Regular Papers, IEEE Transactions, Volumen 52, issue 2, páginas 350-365. 14 febrero 2005.
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1393167
http://www.coursehero.com/file/4924483/GuoAnFPGAExample/

[29] Syed Shahzad Shah; Saqib Yaqub and Faisal Suleman. " Part one: Viterbi codecs". Self-correcting codes conquer noise .15 Febrero 2001.
ftp://ftp.radionetworkprocessor.com/pub/reed-solomon/viterbi-chameleon.pdf

[30] Documento Actel: "Designing Telecommunications Applications Using Digital Signal Processing Functions with FPGAs". Application Note AC121, 1997 Actel Corporation.
http://www.actel.com/documents/Telecom_DigSig_AN.pdf

2.8.5 Intercambio de registros y método traceback.

[31] Herbert Dawid, Olaf J. Joeressen and Heinrich Meyr. "Chapter 17 Viterbi Decoders: High Performance Algorithms and Architectures". Páginas 24-32.
http://www.eecs.berkeley.edu/newton/Classes/EE290sp99/lectures/ee290aSp996_1/vit_chap17.pdf

[32] T.K. Truong; Ming-Tang Shih; Irving S. Reed and E.H. Satorius. "A VLSI Design for a Trace-Back Viterbi Decoder". IEEE Transactions on Communications, Vol 40, Nº3, March 1992.
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=135732

http://www.mendeley.com/research/vlsi-design-traceback-viterbi-decoder-8/

http://biblioteca.universia.net/html_bura/ficha/params/title/vlsi-design-for-trace-back-viterbi-decoder/id/43055355.html

[33] Hui-Ling Lou. "Implementing the Viterbi Algorithm". Fundamentals and real-time issues for processor designers. IEEE Signal Proccesing Magazine. September 1995.
http://www.mendeley.com/research/implementing-the-viterbi-algorithm-1/

[34] Juli Ordeix; Pere Martí; Moisés Serra y Jordi Carrabina. "Arquitectura de un decodificador convolucional a partir del algoritmo de Viterbi". Departamento de Informática, ETSE, Universidad Autónoma de Barcelona. Apartados 3.2 y 3.3.

2.8.6 Otras referencias.

[35] Irina E. Bocharova; Florian Hug; Rolf Johannesson and Boris D. Kudryashov. "An Analytic Expression for the Exact Bit Error Probability for Viterbi Decoding of Convolutional Codes". IEEE Transactions on Information Theory, November 2011. http://arxiv.org/PS_cache/arxiv/pdf/1111/1111.3820v1.pdf

[36] Andrei Vityaev and Paul H. Siegel. "On Viterbi Detector Path Metric Differences". IEEE Transactions on Communications, Vol. 46, Nº 12, pp 1549-1554. December 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.5242

[37] Miguel Calvo Ramón y Ramón Martínez Rodríguez-Osorio. "Codificación de canal. Codificación por bloques". ETSIT, Escuela Técnica Superior de Ingenieros de Telecomunicación, Universidad Politécnica de Madrid. Comunicaciones por Satélite, curso 2008-2009.
http://www.gr.ssr.upm.es/docencia/grado/csat/material/CSA08-5-CodificacionBloques.pdf

[38] "Recordatorio técnicas básicas y UMTS". UPM, Universidad Politécnica de Madrid. DIT, Departamento de Ingeniería de Sistemas Telemáticos. Redes y Servicios de Radio, curso 2005-2006.
http://ocw.upm.es/ingenieria-telematica/redes-y-servicios-de-radio/contenidos/rsrd-OCW/web-05-06/recordatorio-tecnicas-basicas-y-umts-05-06-Parte1.pdf

[39] Belén Lara Aznar. "Codificación de Datos. Nuevas Tecnologías en Comunicaciones Móviles". I+E Revista Digital Investigación y Educación. Número 10, ISSN 1696-7208 . Septiembre 2004.
http://www.csi-csif.es/andalucia/modules/mod_sevilla/archivos/revistaense/n10/Datos.PDF

[40] Héctor Abarca A. "Corrección de errores". Sistemas de Comunicación de Datos II.
h**abarca**.files.wordpress.com/2008/04/scdii_errores_3.ppt

[41] Andrew James Viterbi. "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm". Publicado en: Information Theory, IEEE Transactions on. Pages 260-269, Volume 13, Issue 2. Abril 1967.
http://ebookbrowse.com/error-bounds-for-convolutional-codes-and-an-asymptotically-optimum-decoding-algorithm-pdf-d62569248

http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1054010

[42] Peter Elias. "Error-free Coding". IEEE transactions on Information Theory. Pages 29-37, Volume 4, Issue 4. Septiembe 1954.
http://dspace.mit.edu/bitstream/handle/1721.1/4795/RLE-TR-285-14266170.pdf?sequence=1

http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1057464

http://www.mendeley.com/research/errorfree-coding/#

[43] Richard E. Blahut. “Algebraic Codes for Data Transmission”. Cambridge University Press. Páginas 278-282. 10 Marzo de 2003.

[44] Michael Francis. "Viterbi Decoder Block Decoding – Trellis Termination and Tail Biting". Documentación de Xilinx, XAPP551 (v2.0), 30 de Julio de 2010. http://www.xilinx.com/support/documentation/application_notes/xapp551.pdf

[45] Syed Shahzad Shah; Saqib Yaqub and Faisal Suleman. " Part one: Viterbi codecs". Self-correcting codes conquer noise .15 Febrero 2001. ftp://ftp.radionetworkprocessor.com/pub/reed-solomon/viterbi-chameleon.pdf

[46] Hui-Ling Lou. "Implementing the Viterbi Algorithm". Fundamentals and real-time issues for processor designers. IEEE Signal Proccesing Magazine. Page 47. September 1995. http://www.mendeley.com/research/implementing-the-viterbi-algorithm-1/

[47] Shung C.B.; Siegel P.H.; Ungerboeck G. and Thapar H.K. "VLSI architectures for metric normalization in the Viterbi algorithm". In: 1990 *IEEE International Conference on Communications,* Atlanta, Georgia. Pages 1723-1728, volume 4. 16-19 April 1990. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=117356

CAPÍTULO 3

3. IMPLEMENTACIÓN CODIFICADOR CONVOLUCIONAL: EncoderK7.vhd

3.1 Características.

Implementamos un codificador convolucional, *EncoderK7.vhd*, con estos parámetros:

- Constraint length K=7. Definida como la longitud en bits del polinomio generador. El codificador tiene 6 registros de memoria y $2^{K-1} = 64$ estados.
- Code rate R = 1/2, (tasa = 1/2).
- Polinomio generador 171, 133 (octal). 1111001 y 1011011. $Dout_1$ corresponde al 171 y $Dout_0$ al 133.
- Lo desarrollamos íntegramente en código VHDL multiplataforma.
- Latencia 1 T_{CLK} entre EnableIn y EnableOut. Y entre Din y Dout(1:0). Esta latencia se debe a que utilizamos salidas registradas.
- Período de proceso de un dato 1 T_{CLK}. Cada bit de entrada llega con frecuencia máxima F_{CLK}. Y en la salida hay un dato de dos bits con frecuencia F_{CLK}.
- Puede codificar una cadena de bits continua o dividida en tramas. Los bits de entrada pueden llegar de manera continua o en pulsos.

El polinomio generador es el óptimo para un codificador R = 1/2 y K = 7, porque es el que maximiza la distancia mínima de Hamming (d_{min} ó d_{free}). En [8], [9], se indica cuáles son los polinomios más adecuados para cada tipo de codificador.

Por este motivo este codificador se emplea en múltiples aplicaciones FEC. Es el estándar industrial para codificadores R = 1/2 y constraint length = 7. Es compatible con los siguientes estándares: Q1900, DVB, IEEE802.11a, IEEE802.16a, HiperAccess, HiperMAN, INTELSAT IESS-308/309, referencias [6], [7].

El codificador convolucional es un elemento imprescindible en este proyecto. Se utiliza en los simuladores, descritos en el *tema 6*, que nos permiten verificar que el decodificador Viterbi funciona correctamente. En estos simuladores se genera una cadena de bits aleatorios, que son la entrada al codificador. A los bits de salida del codificador se les aplica el ruido del canal, y el resultado llega a la entrada del Viterbi. Para finalizar, se compara la salida del decodificador con la entrada al codificador.

La manera más sencilla de implementar un codificador sería utilizar un IP core que ya lo implemente. Por ejemplo, se puede usar el de Xilinx, referencias [1], [2] y [3]. Pero nosotros no utilizamos este IP core, ni ningún otro, porque un objetivo fundamental de nuestro proyecto es implementar un código multiplataforma. En el *apartado 1.7* detallamos porqué en un código multiplataforma no se pueden utilizar IP cores.

Por este motivo desarrollamos nosotros mismos el codificador convolucional. Su código, al igual que el del resto de elementos que desarrollamos en este proyecto, es multiplataforma. De manera que no sólo cumpliremos el objetivo inicial de un

decodificador Viterbi multiplataforma. Además, todos los simuladores que desarrollamos en VHDL también tendrán esta característica.

El codificador se utiliza en todos los simuladores descritos en el *tema 6*, tanto los de VHDL como los de System Generator. En el caso de System Generator, el codificador lo implementamos con una caja negra, en la que importamos el código del *EncoderK7.vhd*. Debemos tener esto en cuenta al desarrollar el código, porque debe cumplir los requerimientos necesarios para poder utilizarlo en cajas negras de System Generator. Estos requerimientos están en las páginas 64 a 71 de la referencia [3].

3.2 Interfaz entrada/salida. Uso del codificador.

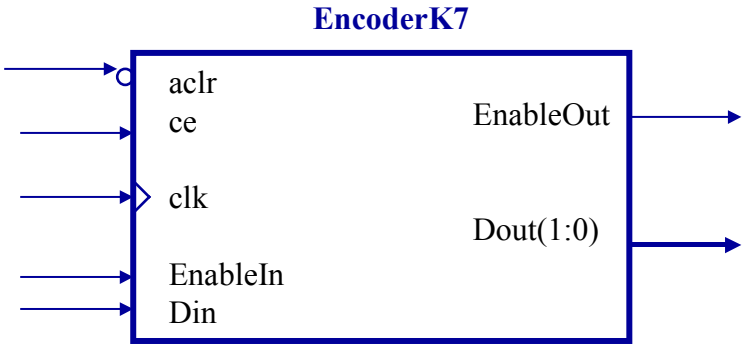
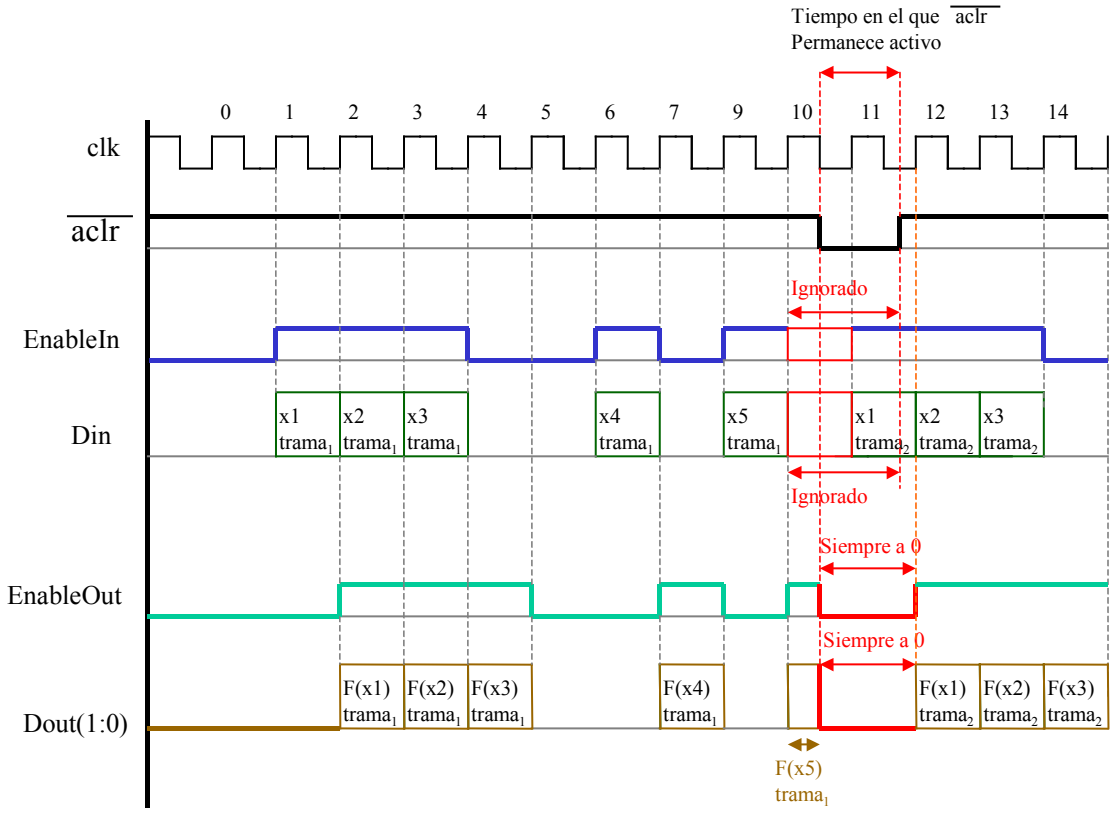
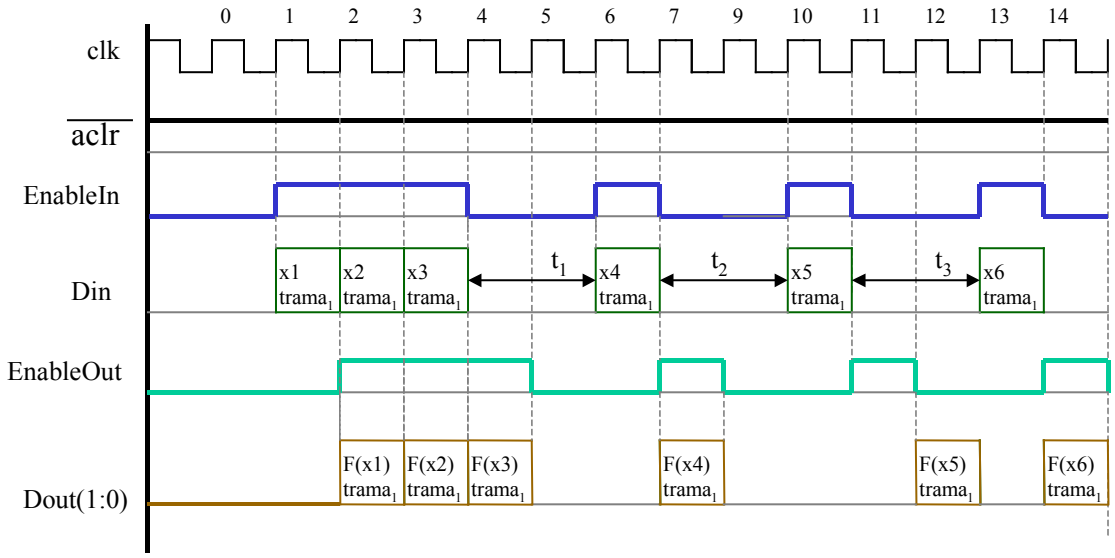


Figura 3.1: Interfaz del codificador: EncoderK7.vhd.



Cronograma 3.1: Codificador UC3M, EncoderK7.vhd, efecto $\overline{\text{aclr}}$.



Cronograma 3.2: Codificador UC3M, codificación continua y en pulsos.

Basándonos en los cronogramas explicamos los detalles del funcionamiento:

- El período de proceso de un dato es de 1 T_{CLK}.
- Los bits de entrada pueden llegar a Din de dos maneras diferentes:
  1. De manera continua, con frecuencia F_{CLK}.
  2. En pulsos, se produce un pulso de duración T_{CLK} en el que llega un bit. Y entre dos pulsos consecutivos transcurre un tiempo $x*T_{CLK}$. x puede tener cualquier valor entero, en el *cronograma 3.2* el tiempo $x*T_{CLK}$ se representa por t_1, t_2 y t_3. Todos estos tiempos pueden tener cualquier valor.
- Sí en el flanco activo de reloj EnableIn está activado, a nivel alto, entonces el codificador lee el bit que haya en Din en ese instante. El resultado de la codificación de ese bit estará disponible en Dout(1:0) en el siguiente ciclo de reloj.
- EnableOut es la señal EnableIn retrasada 1 T_{CLK}. EnableOut[n] = EnableIn[n-1]
- Entre Din y Dout(1:0) también hay un T_{CLK} de latencia y tenemos:
  - Para obtener $Dout_1$ hay que aplicar el polinomio $1+x+x^2+x^3+x^6$ a Din y añadir un retardo de un T_{CLK}. El polinomio se representa como 1111001 en binario, 171 en octal. Y en la salida se obtiene:
    $Dout_1[n]=Din[n-1]+Din[n-2]+Din[n-3]+Din[n-4]+Din[n-7]$.
  - Para obtener $Dout_0$ hay que aplicar el polinomio $1+x^2+x^3+x^5+x^6$ a Din y añadir un retardo de un T_{CLK}. El polinomio se representa como 1011011 en binario, 133 en octal. Y en la salida se obtiene:
    $Dout_0[n]=Din[n-1]+Din[n-3]+Din[n-4]+Din[n-6]+Din[n-7]$.

- $\overline{\text{aclr}}$ es un reset asíncrono que fuerza todas las salidas a '0'.

- Si la cadena de entrada es continua, entonces sólo se debe activar $\overline{\text{aclr}}$ una vez, durante al menos un T_{CLK}, antes de que llegue el primer bit de la cadena.

- Si la cadena de entrada está dividida en tramas, entonces hay que activar $\overline{\text{aclr}}$ durante al menos un T_{CLK}, antes de que llegue el primer bit de cada una de las tramas.

- EnableIn actúa como un clock enable. Permite pausar la codificación de una trama o cadena de bits. Si mientras se está recibiendo una trama de entrada se desactiva EnableIn, la codificación se pausa. Al volver a activar EnableIn la codificación se reanuda, pasando a codificar el siguiente bit de la trama de entrada. Para conseguir esto, al desactivar EnableIn, los 6 registros de memoria del módulo conservan su valor. De manera que en el estado de pausa se conservan los valores desde Din[n-1] hasta Din[n-6], que había en el último instante en el que EnableIn estuvo activo.

- Esta característica permite que los bits de la cadena o trama de entrada lleguen de manera continua con frecuencia F_{CLK}, o en forma de pulsos. Si llegan de manera continua, EnableIn estará permanentemente activo. Si lo hacen en forma de pulsos, EnableIn se mantiene activo durante un tiempo T_{CLK}, momento en el que llega un bit a Din. Y permanece desactivado un tiempo $x*T_{CLK}$, momento en el que no llegan bits a Din.

3.2.1 Definición de puertos.

Tabla 3.1: Puertos EncoderK7.vhd		
Puerto	Dirección	Descripción
Clk	Entrada	Reloj del sistema.
$\overline{\text{aclr}}$	Entrada	Reset asíncrono, activo a nivel bajo. Pone todos los puertos de salida, las señales internas y los registros internos a cero. Si se activa mientras el módulo está codificando una trama, ya no se podrá reanudar la codificación de la trama porque se borra la memoria. Obligatorio activarlo durante al menos 1 T_{CLK} antes del inicio de cada trama.
Ce	Entrada	Es el clock enable. Se utiliza para compatibilizar el diseño con System Generator. Si ce = '1', no afecta a la codificación. Si ce = '0', entonces la codificación se para, pero almacena su estado. De manera que cuando ce vuelva a ser uno, la codificación se reanuda correctamente desde el estado almacenado. Es equivalente a una pausa y se puede activar y desactivar en cualquier momento incluso en medio de una trama.
EnableIn	Entrada	Debe activarse a nivel alto siempre que en Din haya un bit válido. Su activación indica al codificador que debe leer el bit presente en Din. Si EnableIn vale '0', entonces el codificador no lee el bit presente en Din. Este es un estado de pausa, el módulo no codifica el bit de entrada, pero conserva los valores de sus registros internos.
Din	Entrada	Bit de entrada al codificador.
EnableOut	Salida	Si esta señal está activa, a nivel alto, entonces $Dout_1$ y $Dout_0$ contienen un valor válido. Si EnableOut es '0', entonces no hay valores válidos en $Dout_1$ ni en $Dout_0$.
Dout(1:0)	Salida	Bits de salida del codificador. $Dout_1$ corresponde a aplicar el polinomio $1+x+x^2+x^3+x^6$ a Din y añadir un retardo de un T_{CLK}. Polinomio 1111001 en binario, 171 en octal. $Dout_0$ corresponde a aplicar el polinomio $1+x^2+x^3+x^5+x^6$ a Din y añadir un retardo de un T_{CLK}. Polinomio 1011011 en binario, 133 en octal.

3.3 Arquitectura del codificador.

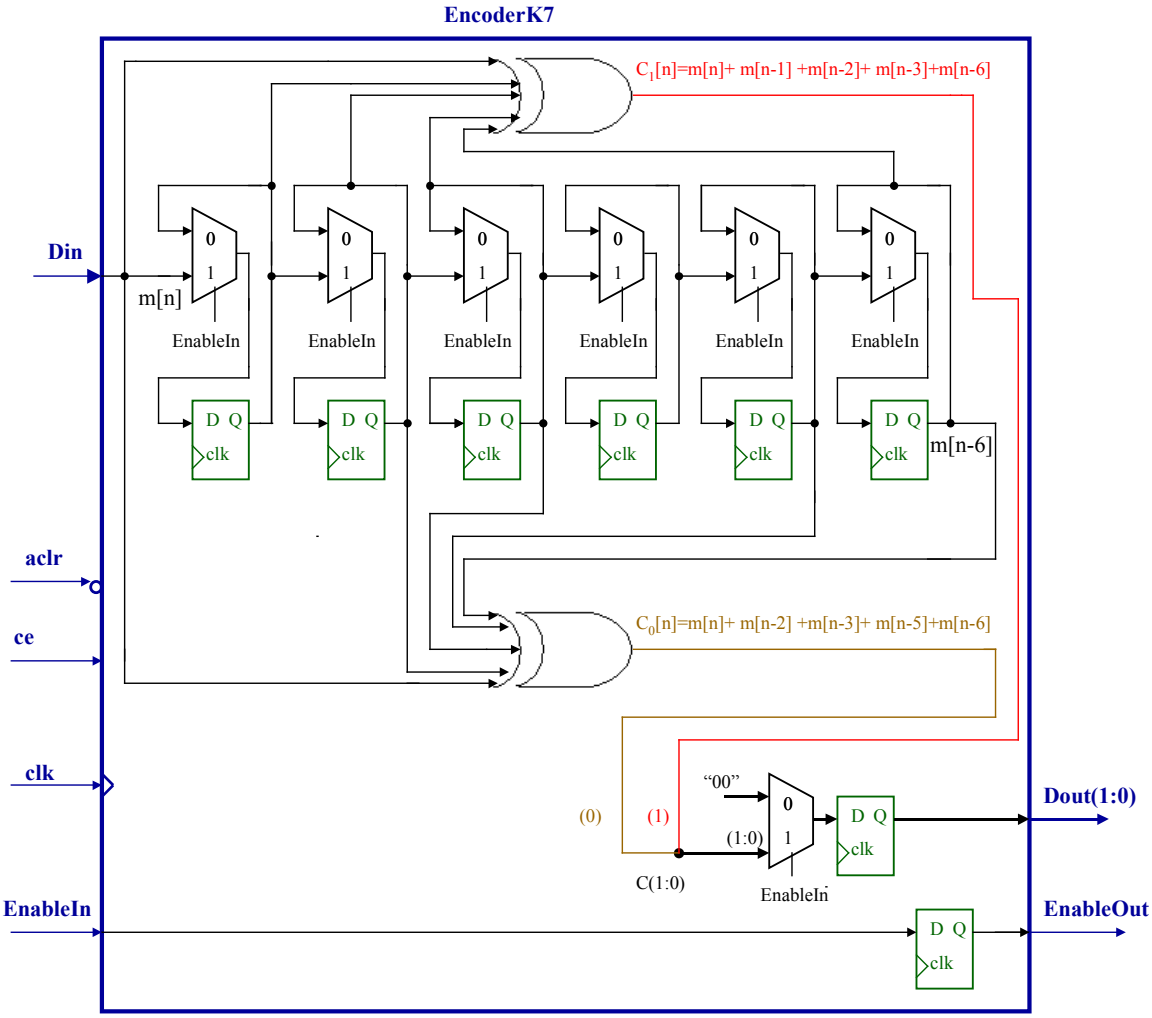
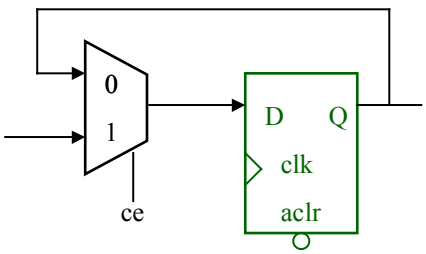
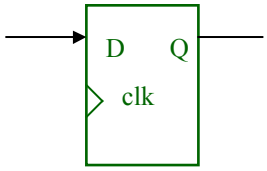


Figura 3.2: Arquitectura codificador convolucional UC3M: EncoderK7.vhd.

Nota 3.1: Todos los registros del módulo tienen las señales ce y $\overline{\text{aclr}}$. Sin embargo no las incluimos en la *figura 3.2* para evitar redundancias en la representación gráfica.



Estructura exacta de cada uno de los registros del codificador.



Representación simplificada en la figura 3.2.

Figura 3.3: Estructura exacta de los registros.

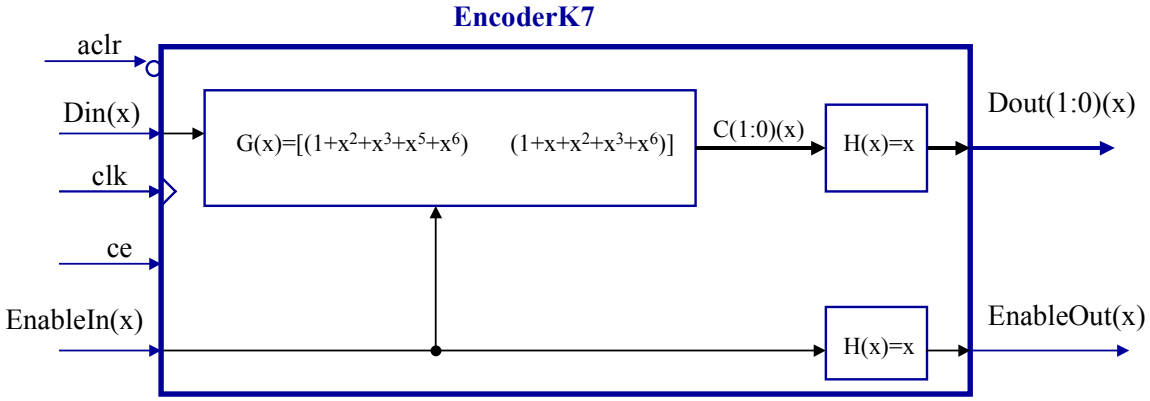
La arquitectura consiste en unos registros de desplazamiento, *shift registers*, conectados con sumadores base 2, que representan un polinomio concreto. Para diseñar esta arquitectura nos basamos en las referencias citadas en *2.8.1A* y en el *apartado 2.4*.

En C(1:0) tenemos la codificación de Din con el polinomio especificado. Sin embargo, no podemos utilizar esta señal directamente como salida, es imprescindible añadir un registro a continuación, para obtener una salida registrada.

C(1:0) es una señal obtenida a partir de lógica combinacional y por tanto tiene glitches. Entonces es necesario registrar esa señal, de manera que obtenemos Dout(1.0), que no tiene glitches por ser la salida de un registro. Esto es algo básico y fundamental en el diseño síncrono, puede consultarse en las referencias [4] y [5]. En caso de no registrar la salida y utilizar directamente C(1:0), cometeríamos un fallo grave de diseño. Porque la salida tendría glitches, y el sistema sería combinacional en vez de síncrono.

Registrar la salida es una técnica habitual en los módulos codificadores convolucionales. Esto se puede apreciar, por ejemplo, en la figura 8 del datasheet del codificador convolucional de Xilinx, [1].

Además cumplimos todas las reglas de diseño síncrono y el flujo de diseño del *apartado 1.8*. De esta manera nos aseguramos de que hemos cumplido todos los pasos que requiere la implementación de un sistema digital multiplataforma.



A continuación mostramos la representación matemática exacta del codificador:

Figura 3.4: Función de transferencia del codificador convolucional.

Las funciones de transferencia en forma polinómica son:
$Dout(x)=Din(x)G(x)H(x)=C(x)*x.$; $Dout_1(x)=Din(x)g_{12}(x)$; $Dout_0(x)=Din(x)g_{11}(x)$

$C_1(x)=Din(x)*(1+x+x^2+x^3+x^6)$.
$C_0(x)=Din(x)*(1+x^2+x^3+x^5+x^6)$.
$Dout_1(x) = C_1(x)*x = Din(x)*(1+x+x^2+x^3+x^6)*x = Din(x)*(x+x^2+x^3+x^4+x^7)$.
$Dout_0(x) = C_0(x)*x = Din(x)*(1+x^2+x^3+x^5+x^6)*x = Din(x)*(x+x^3+x^4+x^6+x^7)$.

$EnableOut(x)=EnableIn(x)*x$.

Las funciones de transferencia en forma de tiempo discreto son:

$C_1[n] = Din[n]+Din[n-1]+Din[n-2]+Din[n-3]+Din[n-6]$.
$C_0[n] = Din[n]+Din[n-2]+Din[n-3]+Din[n-5]+Din[n-6]$.
$Dout_1[n] = C_1[n-1] = Din[n-1]+Din[n-2]+Din[n-3]+Din[n-4]+Din[n-7]$.
$Dout_0[n] = C_0[n-1] = Din[n-1]+Din[n-3]+Din[n-4]+Din[n-6]+Din[n-7]$.

EnableOut[n]=EnableIn[n-1].

3.4 Verificación de que funciona correctamente.

3.4.1 Pasos a seguir.

Tras implementar el codificador, sintetizarlo y simularlo, nos aseguramos de que hemos seguido adecuadamente todos los pasos que definíamos en el *apartado 1.8*.

Con las simulaciones nos aseguramos de que el funcionamiento general es el adecuado, es decir, no hay errores en los puertos EnableIn, EnableOut, $\overline{aclr}$, clk y ce. También nos sirve para asegurarnos de que la tasa R es efectivamente de 1/2. Ante un bit en Din se obtienen 2 en Dout(1:0) y con un T_{CLK} de latencia.

Sin embargo en esta ocasión esta simulación no es suficiente. Puesto que falta asegurar, que la salida Dout(1:0) es el resultado de codificar convolucionalmente la entrada Din con los polinomios 171 y 133. Para ello es obligatorio realizar las siguientes pruebas:

1. Comparar nuestro codificador con otro que usaremos como referencia. Lógicamente, los dos deben tener las mismas características: K=7; R=1/2 y polinomios 171, 133. También deben tener los mismos puertos: EnableIn, EnableOut, Din, Dout(1:0), clk y ce.

2. Debemos tener la certeza absoluta de que el codificador de referencia es funcionalmente correcto.

3. Es necesario generar una cadena aleatoria de bits y que ambos módulos codifiquen la misma cadena. *Figura 3.4.*

4. Hay que hacer medidas con varias cadenas de bits y cada una de ellas debe ser suficientemente larga, varios millones.

5. A continuación se comparan las salidas de los dos codificadores. Deben ser exactamente iguales, no se admite ni un sólo bit de diferencia entre ambas salidas. DoutEncoderK7(1:0)[n]=DoutEncoderReferencia(1:0)[n], para todo n. Ver *figura 3.4*.

6. Si se cumplen las 5 condiciones anteriores, entonces podemos verificar con un 100 % de seguridad que nuestro codificador es funcionalmente correcto.

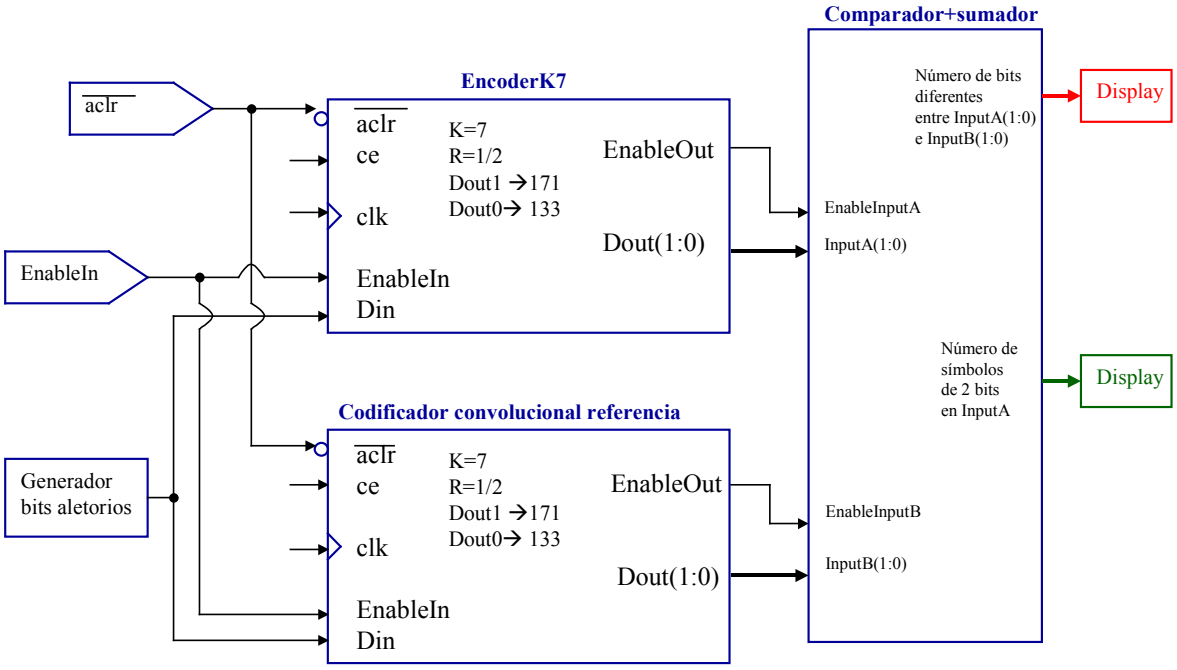


Figura 3.5: Sistema para verificar que EncoderK7.vhd es funcionalmente correcto.

3.4.2 Simulación usando como referencia ConvolutionalEncoder.m

Tras implementar el *EncoderK7.vhd*, desarrollamos un codificador convolucional con código Matlab, *ConvolutionalEncoder.m*, y lo utilizamos como referencia. Entonces realizamos las pruebas del apartado anterior, y comprobamos que ambos módulos obtienen la misma salida. De manera que podemos asegurar que el *EncoderK7.vhd* funciona exactamente igual que el bloque de Matlab.

Sin embargo esta comprobación no es suficiente, porque no tenemos la certeza absoluta de que el codificador de referencia, el implementado en Matlab, es correcto. Mediante esta prueba nos aseguramos de que ambos bloques funcionan igual, pero no podemos asegurar que funcionen bien, porque hemos desarrollado nosotros los dos. De manera que podría darse el caso de que hubiésemos cometido el mismo error en ambos, de manera que la salida de los dos sería la misma, pero errónea.

Otro inconveniente añadido es que el proceso de pruebas es muy lento, porque no hemos desarrollado un simulador que incluya todos los módulos que intervienen en una prueba. En cada prueba hay que realizar varios pasos. Además en cada paso hay que ejecutar un módulo, de Matlab o de VHDL, y utilizar el fichero generado como entrada para el siguiente paso:

1. Generar la cadena de bits aleatorios, para ello se emplea *GenFichAleatorio.m*

2. Codificar la cadena aleatoria mediante los dos codificadores, su salida se escribirá en un fichero.

3. Mediante *ComparaFicheros.m* verificamos que los dos ficheros de salida de ambos codificadores son exactamente iguales.

3.4.3 Verificación comparando con Xilinx IP core convolutional encoder.

En el apartado anterior hemos visto como el primer intento que hicimos para verificar el codificador no fue suficiente. De manera que desarrollamos un simulador en System Generator: *ComparaEncoderUC3M_Vs_EncoderXilinx.mdl.* Con este simulador corregimos todos los problemas que teníamos en el apartado anterior.

- El codificador de referencia es el Xilinx LogiCORE IP convolutional encoder v.3.0, [3]. Por lo que tenemos la certeza absoluta de que funciona correctamente. De manera que solucionamos el mayor inconveniente que teníamos al utilizar el *ConvolutionalEncoder.m* como referencia.

- Otro inconveniente del codificador de Matlab es que el proceso de pruebas es lento. Esto también lo solucionamos, porque en el simulador de System Generator se incluyen todos los módulos necesarios para realizar la verificación. Además el proceso es totalmente automático y lo optimizamos para conseguir la mayor velocidad posible.

En las *figuras 3.6 y 3.7* se aprecian los detalles del simulador. La característica más importante es que incluye dos codificadores: el *EncoderK7.vhd* implementado como una caja negra, y el IP core de Xilinx.

El proceso de simulación consiste en:

1. En el simulador se genera una cadena aleatoria de bits que es la entrada de ambos codificadores. El generador es pseudoaleatorio, por lo que es necesario cambiar su semilla de inicialización antes de lanzar una nueva simulación.

2. *ComparaEncoderK7vhd_Con_ConvolutionalEncoderv3_0.m* suma el número de bits diferentes entre las salidas de ambos codificadores. También obtiene el número total de símbolos, (de 2 bits), codificados.

3. Los resultados se muestran en dos displays, que se actualizan instantáneamente en tiempo real.

El proceso es totalmente automático, lo único que hay que hacer es cambiar la semilla de inicialización del generador pseudoaleatorio, lanzar la simulación y esperar los resultados.

Una vez finalizada la implementación del simulador aprovechamos sus ventajas para realizar múltiples pruebas. Cada una de ellas con cadenas aleatorias de varios millones de bits.

En todas las pruebas el resultado ha sido el esperado, no ha habido ni un sólo bit de diferencia en la salida de los dos bloques. **Por tanto podemos concluir con un 100% de seguridad, que hemos implementado un codificador, EncoderK7, que es funcionalmente correcto.**

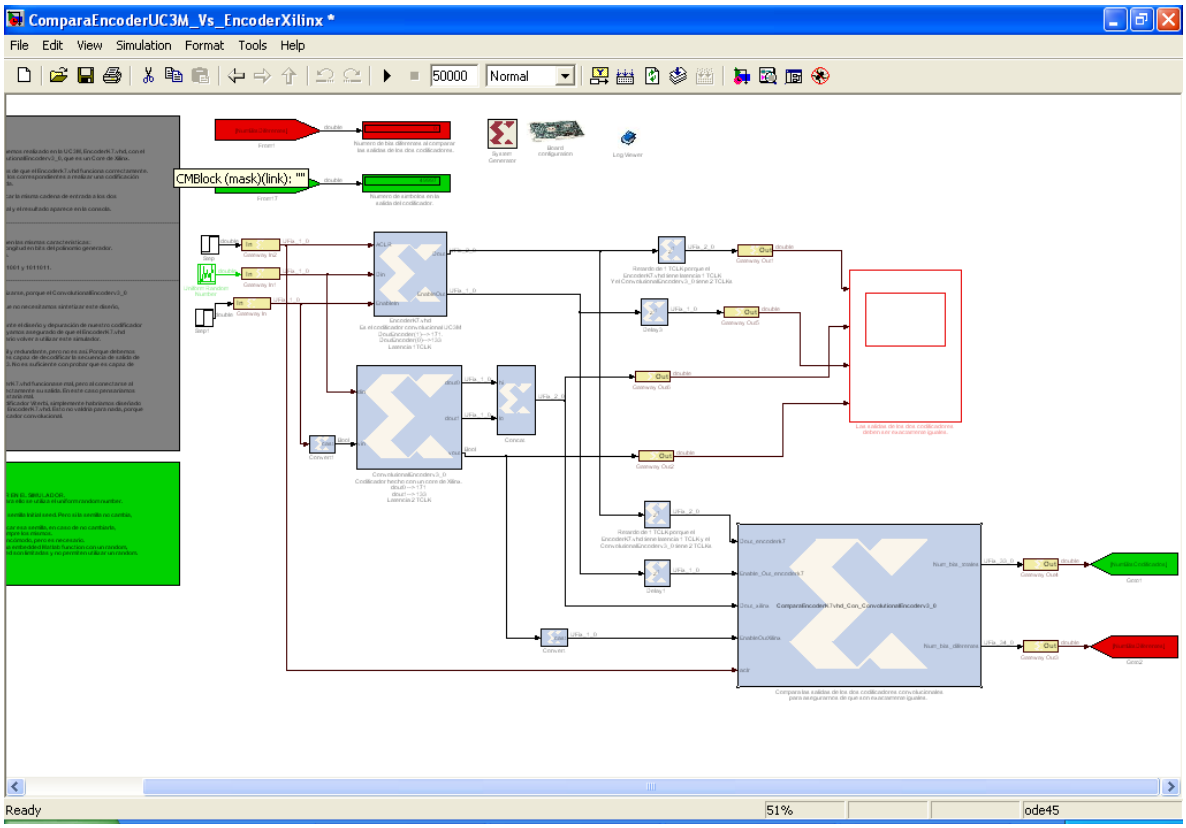


Figura 3.6: Esquema completo de ComparaEncoderUC3M_Vs_EncoderXilinx.mdl.

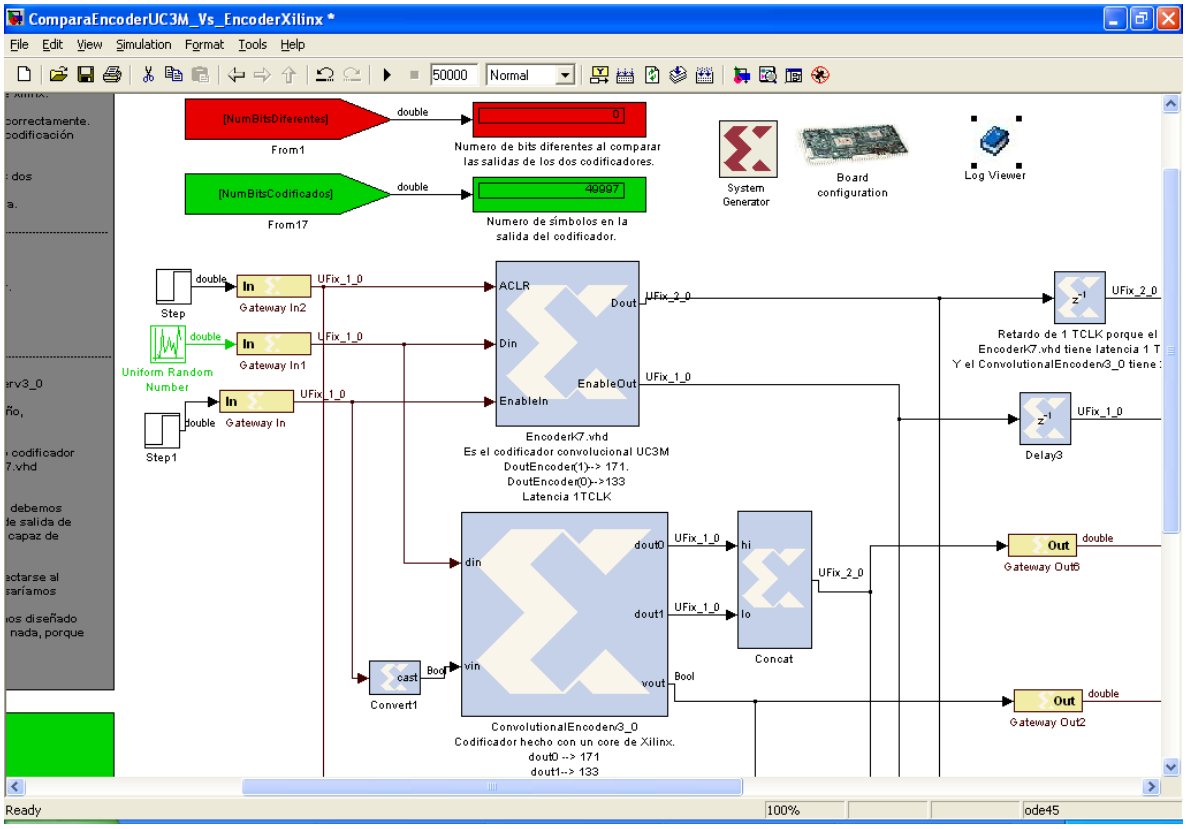


Figura 3.7: Detalle del simulador ComparaEncoderUC3M_Vs_EncoderXilinx.mdl.

3.5 Resultados síntesis. Comparativa con el IP core de Xilinx.

En el apartado anterior hemos verificado que nuestro codificador funciona correctamente y se comporta exactamente igual que el IP core de Xilinx. El siguiente paso, consiste en comparar nuestro módulo con el de Xilinx, en términos de área y velocidad. Así nos aseguraremos de que nuestro diseño es correcto, y está optimizado en área y frecuencia máxima. Los resultados se muestran en la *tablas 3.2 y 3.3*.

En todos los casos, la síntesis y la implementación de los codificadores, la realizamos sobre la FPGA que disponemos en el laboratorio: Virtex 4 xc4vsx55 –10ff1148.

El IP convolutional encoder de Xilinx no requiere licencia, por lo que podemos implementarlo sin problemas. Además implementaremos 2 versiones diferentes, la v4.0 y la v7.0, y compararemos ambas con el codificador UC3M.

Tabla 3.2: Síntesis codificador convolucional UC3M y Xilinx					
Design Summary					
	Codificador UC3M EncoderK7.vhd			Codificador Xilinx IP convolutional encoder	
Logic Utilization	Used	Available	Utilization	Used	
				V4.0	V7.0
Number of Slices	6	24576	0%	9	8
Number of Slice Flip Flops	8	49152	0%	11	11
Number of 4 input LUTs	5	49152	0%	16	15
Number of bonded IOBs	8	640	1%	8	8
IOB Flip Flops	1				
Number of GCLKs	1	32	3%	1	1
Timing Summary					
	EncoderK7.vhd			v4.0	v7.0
Minimum period (ns)	2,099 ns			3,298 ns	2,051 ns
Maximum Frequency (MHz)	476,14 MHz			274,28 MHz	487,6
Minimum input arrival time before clock	3,621 ns			3,298 ns	3,035 ns
Maximum output required time after clock	4,851 ns			5,81 ns	4,677 ns;
Maximum combinational path delay	No path found				

Tabla 3.3: Map codificador convolucional UC3M y Xilinx.					
Design Summary					
	Codificador UC3M EncoderK7.vhd			Codificador Xilinx IP convolutional encoder	
				Used	
Logic Utilization	Used	Available	Utilization	v4.0	v7.0
Number of Slice Flip Flops	5	49152	1%	11	10
Number of 4 input LUTs	5	49152	1%	16	15
Number of occupied slices	6	24576	1%	10	9
Number of Slices containing only related logic	6	6	100%	10	9
Number of Slices containing unrelated logic	0	6	0%	0	0
Total Number 4 input LUTs	5	49152	1%	16	15
Number of bonded IOBs	8	640	1%	8	8
Number of BUFG/BUFGCTRLs	1	32	3%	1	1
Number used as BUFGs	1			1	1
Number used as BUFGCTRLs	0			0	0
Total equivalent gate count for design	102			184	
Additional JTAG gate count for IOBs	384			384	

Al realizar nuestro diseño lo optimizamos en área y velocidad. Nuestro objetivo inicial era aproximarnos al codificador de Xilinx que tomamos como referencia. Los resultados de las tablas anteriores demuestran que no sólo hemos cumplido el objetivo inicial, sino que hemos superado ampliamente nuestras expectativas. Porque el EncoderK7 no sólo es equivalente al IP de Xilinx, sino que lo mejora en prestaciones. Las características más importantes de los 3 codificadores las resumimos a continuación:

- El *EncoderK7.vhd* admite una frecuencia máxima de reloj mucho mayor que el codificador de Xilinx v4: 476,14 MHz, frente a 274,28 MHz. Esta es sin duda la mayor ventaja de nuestro módulo frente al de Xilinx.

- El IP Xilinx v7 está mucho más optimizado en velocidad que el v4, alcanzando los 487,6 MHz, una cifra muy parecida a la que se obtiene con nuestro codificador. En este caso el *EncoderK7.vhd* no aporta ventajas frente al de Xilinx, pero se mantiene en los valores similares de frecuencia máxima.

- Nuestro módulo emplea menos área que las 2 versiones de Xilinx. Además en algunos parámetros esta mejora es muy significativa, por ejemplo:

    - El EncoderK7 emplea 5 LUTs de 4 entradas. Mientras que el Xilinx v4 utiliza 16, y el v7 15. En todos los casos sucede tanto en síntesis como en map. La reducción del área es del orden del 66 %.

    - El EncoderK7 ocupa 6 slice flip flops, en síntesis. En cambio el v4 emplea 9 y el v7 emplea 8. Reducción del 33 % y del 25%.

    - El EncoderK7 utliza 5 slice flip flps, en map, mientras que el v4 ocupa 11 y el v7 10. Reducción del área alrededor del 50%.

3.6 Referencias.

[1] Xilinx: "LogiCORE IP Convolutional Encoder v7.0". DS248 March 1, 2011.
http://www.xilinx.com/support/documentation/ip_documentation/convolution_ds248.pdf

[2] Página Web del Convolutional Encoder v7.0 de Xilinx:
http://www.xilinx.com/products/intellectual-property/Convolutional_Encoder.htm

[3] Documento Xilinx: "System Generator for DSP. Reference Guide". UG638, v13.3, October 19, 2011. Páginas 450-451.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sysgen_ref.pdf

[4] Miguel Ángel Freire Rubio. "Diseño Síncrono de Circuitos Digitales". Ingeniería Técnica de Telecomunicación, EUITT, Universidad Politécnica de Madrid, septiembre 2008.
http://www.euitt.upm.es/uploaded/464/DS_Sep_2008.pdf

[5] "Electrónica Digital, Tema 3, Diseño Síncrono". Departamento de sistemas electrónicos y de control, EUITT, Universidad Politécnica de Madrid, 2010.
http://www.euitt.upm.es/uploaded/464/teoria/tema3/Tema3.pdf

[6] Documento Xilinx LogiCore: "IEEE802-Compatible Viterbi Decoder v1.1". DS204 November 10, 2004.
http://www.xilinx.com/support/documentation/ip_documentation/viterbi_802.pdf

[7] Xilinx "Viterbi Decoder v7.0 Data Sheet". ds247.pdf, página 3, 1 Marzo 2011:
http://www.xilinx.com/support/documentation/ip_documentation/viterbi_ds247.pdf

[8] Christian B. Schlegel and Lance C. Pérez. "Trellis and Turbo Coding". IEEE Press Series on Digital & Mobile Communication, pp. 117-120, 2004.

[9] Bernard Skalar. "Digital Communications, Fundamentals and Applications". Prentice Hall PTR , pp. 408-419, segunda edición 21 Enero 2001.

CAPÍTULO 4

4. IMPLEMENTACIÓN DECODIFICADOR OPENCORES: decoderverilog.v

4.1 Objetivos decodificador de Opencores.

En el proyecto desarrollamos 2 decodificadores Viterbi completos:

1. *ViterbiDecoder.vhd*, es el UC3M, lo diseñamos partiendo de cero y con un diseño y arquitectura totalmente propios. No depende nada del módulo de Opencores.

2. *decoderverilog.v*, lo desarrollamos basándonos en el modelo de Opencores. Toda la documentación de este modelo está disponible en [1].

Inicialmente comenzamos a desarrollar el *ViterbiDecoder.vhd*, puesto que es el objetivo del proyecto, desarrollar un decodificador Viterbi. Sin embargo tuvimos que hacer un cambio en la planificación, porque por causas ajenas al proyecto era conveniente tener listo el decodificador en el plazo de tiempo más breve posible.

Nuestro diseño estaba en una fase inicial, por ello consideramos oportuno dejarlo apartado e implementar el decodificador partiendo de un diseño ya hecho. Entonces lo que hicimos fue utilizar el modelo de Opencores e implementar el *decoderverilog.v*. Gracias a esto conseguimos solucionar el problema de los plazos. Después, una vez implementado, simulado y verificado el *decoderverilog.v,* pudimos continuar con el desarrollo de *ViterbiDecoder.vhd*.

Por tanto es importante aclarar que el decodificador de Opencores es solamente un módulo que utilizaremos de manera temporal, para cumplir con los plazos. Al desarrollarlo descubriremos que no es óptimo, tiene errores y no implementa exactamente el algoritmo de Viterbi. Entonces tendrá un rendimiento muy inferior al de los decodificadores Viterbi comerciales. Sin embargo no trataremos de mejorarlo porque sólo se utilizará de manera temporal, hasta que hayamos finalizado el *ViterbiDecoder.vhd*.

ViterbiDecoder.vhd no tendrá los errores del de Opencores, estará optimizado y su funcionamiento será igual al de los decodificadores Viterbi comerciales. Todos implementan el mismo algoritmo, y por eso todos deben funcionar igual.

Existe otra posibilidad que sería mejorar la opción de Opencores corrigiendo todos sus errores. Pero descartamos este camino porque modificar un código ya escrito no es una buena elección. Primero deberíamos comprenderlo perfectamente y después añadirle mejoras y eliminar los errores. Pero en diseños complejos, como el que nos ocupa, esta técnica requiere mucho tiempo, y además si los cambios son importantes, no suelen quedar bien, porque básicamente lo que se hace es ir poniendo parches al diseño. Por este motivo elegimos partir de cero y desarrollar nuestro propio código independiente.

Opencores proporciona el código fuente del decodificador, pero debemos verificar su funcionamiento. Para eso desarrollamos dos simuladores, *TestSimulaciónCompleta.vhd* y *TestLeeFichero.vhd*. Para no desaprovechar el trabajo, estos simuladores que desarrollamos valen para cualquier decodificador, por lo que nos servirán también para el nuestro, *ViterbiDecoder.vhd* y el de Xilinx.

Opencores proporciona un simulador, pero no lo utilizamos porque es muy básico. Así que no nos resulta útil porque la simulación es una parte fundamental. De manera que como valor añadido al proyecto, desarrollamos nosotros mismos un simulador mucho más completo, que nos permitirá verificar el funcionamiento de los decodificadores de una manera mucho más exhaustiva.

El código fuente de este decodificador es Verilog. Como indicamos en el *apartado 1.6,* Verilog es un lenguaje estándar usado para describir sistemas digitales. Y se trata junto a VHDL, del HDL de referencia soportado por la gran mayoría de herramientas de síntesis. Verilog se convierte en norma en 1995, norma IEEE 1364-1995. Esta norma se revisó en 2001, resultando la norma IEEE 1364 –2001.

Debemos conocer la sintaxis de este lenguaje, porque aunque el código fuente se genera automáticamente, examinaremos el código para así conocer la arquitectura del decodificador. Además debemos hacer cambios en el código, ver *apartado 4.5.2*:

- Para conseguir que el código sea compatible con System Generator.
- Para adaptarlo a nuestras especificaciones.
- Para adaptarlo a los puertos de entrada y salida que necesitamos.

Para estudiar la sintaxis del lenguaje Verilog utilizamos las referencias del *apartado 4.8.1*, que incluyen la especificación completa, tutoriales, ejemplos, guías de referencia....

4.2 Pasos seguidos en el proyecto.

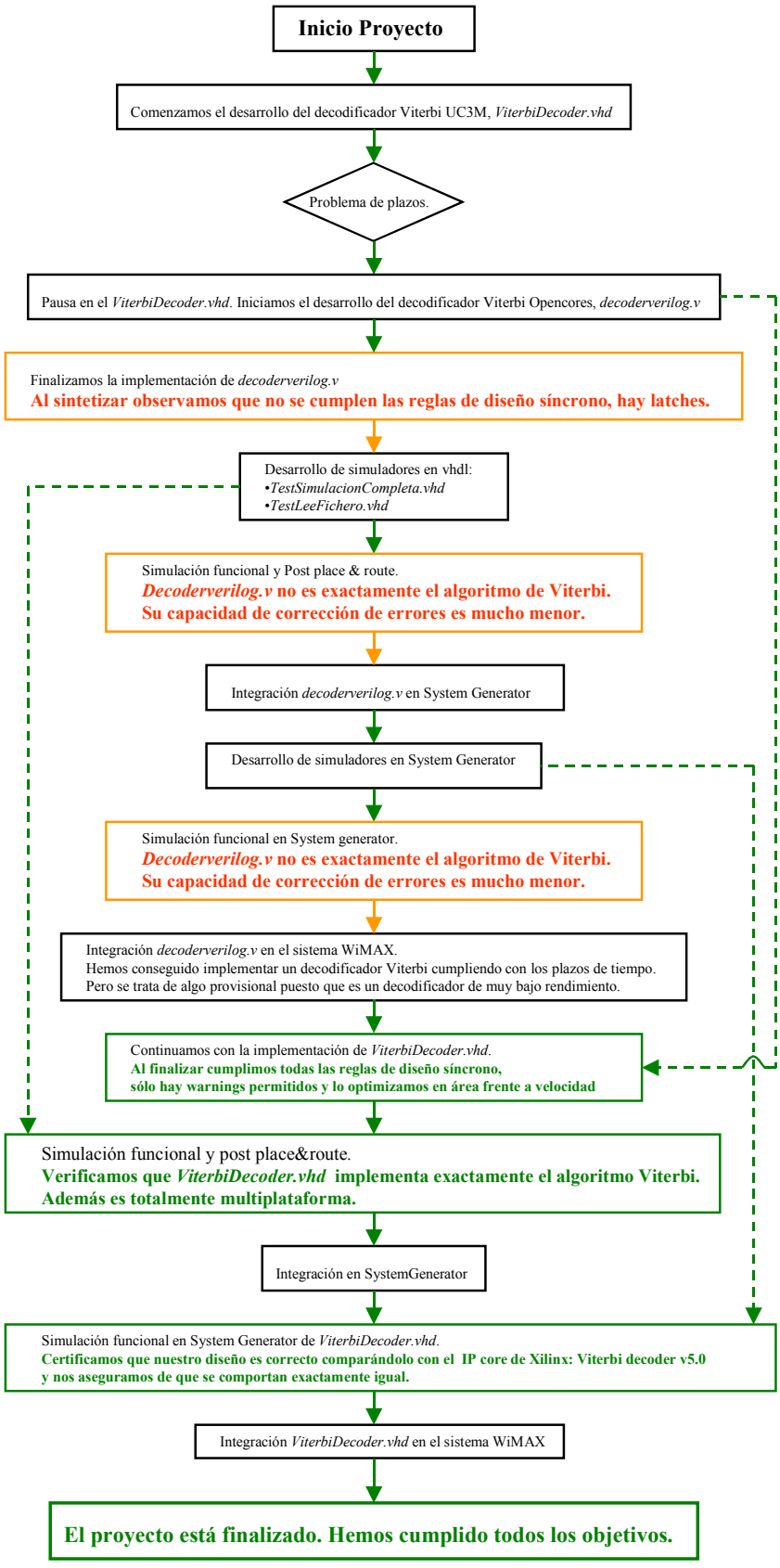


Figura 4.1: Desarrollo del proyecto.

En Internet está disponible un código libre que implementa el algoritmo de Viterbi. Lo proporciona la Web http://opencores.org/. Una página especializada que proporciona módulos hardware codificados en VHDL y Verilog. Más información en [1].

Somos conscientes de que en Internet hay productos buenos, pero también se pueden encontrar multitud de productos que luego no son lo que se espera de ellos. Por eso utilizamos la Web de Opencores, porque se trata de una página seria y podemos confiar en sus diseños.

En la figura anterior se aprecia que la opción de Opencores tiene errores, no es óptima, y su funcionamiento está muy por debajo del de los decodificadores Viterbi comerciales. Todos estos problemas aparecen marcados en rojo en la figura y los explicaremos con detalle en los *apartados: 4.6, 7.4 y 7.7.5*.

Sin embargo en la figura podemos comprobar como en el decodificador UC3M, todos esos errores se han solucionado y su funcionamiento es igual al de los módulos comerciales. Por ello todos los logros que hemos conseguido aparecen marcados en verde.

Al decantarnos por la opción de Opencores, ya sabíamos que no íbamos a conseguir un buen rendimiento. El motivo está claro, los desarrolladores de hardware tienen módulos que implementan el algoritmo Viterbi, pero no son libres. Por ejemplo el IP core Viterbi v5.0 de Xilinx cuesta entre 5000$ y 15.000$. Entonces es lógico pensar que al ser gratuito, el decodificador de Opencores tendrá peores prestaciones que el de Xilinx. No tendría sentido si tuviese un rendimiento equivalente porque entonces no se utilizarían los diseños de pago, los desarrolladores utilizarían las versiones gratuitas. (Toda la documentación del decodificador de Xilinx está en el *apartado 1.9.5*).

Sin embargo, el bajo rendimiento que tiene *decoderverilog.v* no supone ningún problema para el proyecto. Porque este módulo sólo se utilizó de manera temporal, hasta que tuvimos finalizado *ViterbiDecoder.vhd*.

4.3 Características.

Implementamos el decodificador utilizando un generador automático de código que nos proporciona Opencores. Este generador tiene unos parámetros de entrada con los que se indican las propiedades que tendrá el decodificador Viterbi generado automáticamente. En nuestro caso esas propiedades deben ser las que indican las especificaciones en el *apartado 1.4*, y serán las mismas tanto para el *decoderverilog.v* como para el *ViterbiDecoder.vhd*.

Parámetros importantes:

- *Constraint length* K=7. Definida como la longitud en bits del polinomio generador. El codificador tiene (K-1) = 6 registros de memoria. $2^{K-1} = 64$ estados.

- *Decode rate* R = ½, (tasa = 1/2). 2 bits de entrada, 1 de salida.

- Polinomio generador 171, 133 (octal). 1111001 y 1011011.

- *Decoding depth* (profundidad de memoria) = 32. *Nota 4.1*.

- Decodificación dura.

- Período de proceso de un dato en el decodificador = 8 T_{CLKs}. Es el número de T_{CLKs} que necesita el decodificador entre dos datos de entrada consecutivos.
  Este período de proceso indica que el decodificador debe trabajar internamente con una frecuencia 8 veces mayor que la frecuencia con la que llegan datos a su entrada. Por tanto en la entrada habrá un dato nuevo cada 8 T_{CLKs} pero internamente el decodificador trabaja con período T_{CLK}.

- En la entrada el dato debe estar estable durante 8 T_{CLKs}.

- Latencia = 293 T_{CLKs}. $\rightarrow$ (293/8) = 36,625 períodos de proceso, es el número de T_{CLKs} que emplea el decodificador en decodificar un dato.

Nota 4.1: La especificación es de decoding depth = 36, pero no se puede implementar el *decoderverilog.v* con esa profundidad de memoria. Por eso lo desarrollamos con 32 bits, que es el valor más próximo que podemos conseguir. Este cambio supondrá una ligera disminución en la capacidad de corrección de errores, ver resultados en el *apartado 7.7.3*. Pero el decodificador es válido aunque no cumpla esa especificación. No es un fallo, sino simplemente una pequeña disminución del rendimiento.

4.4 Interfaz Entrada/Salida. Uso del decodificador.

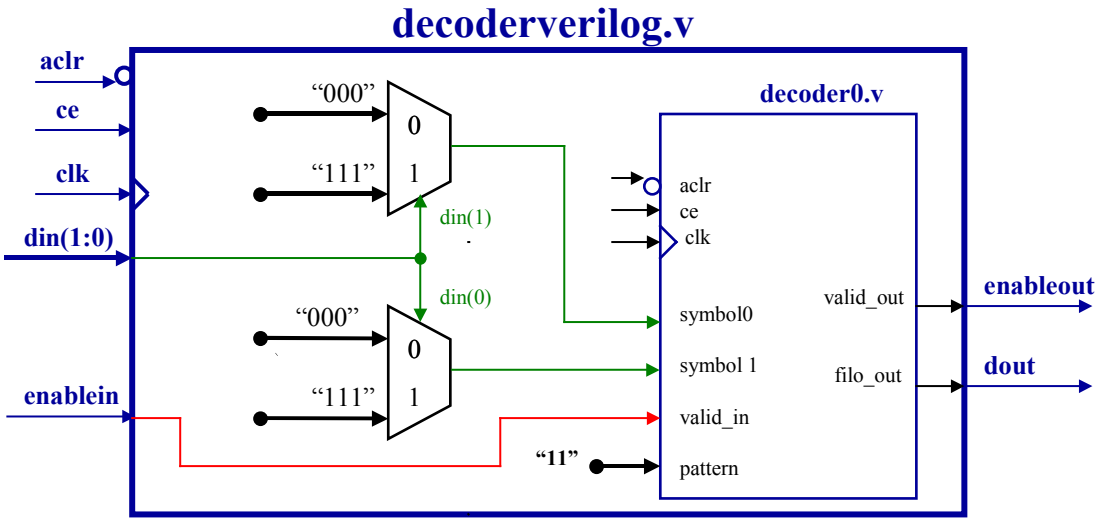
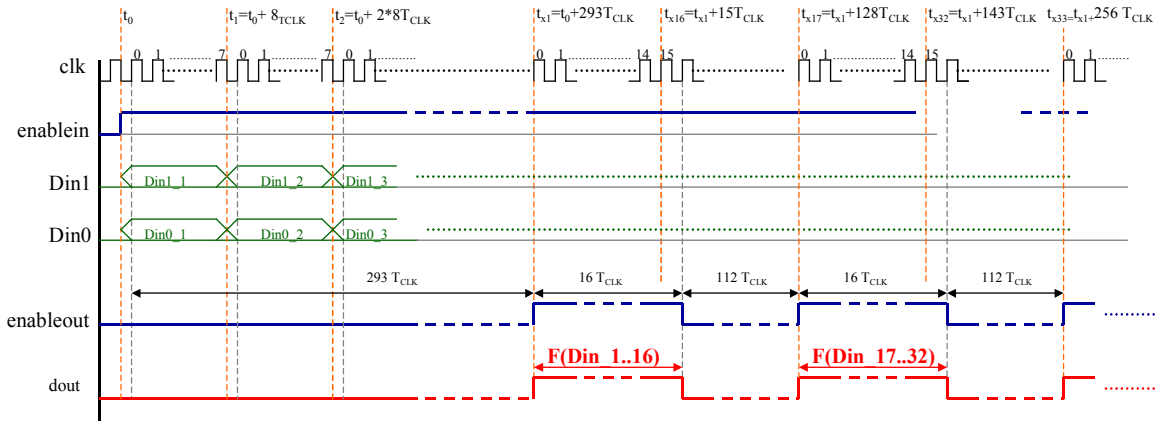
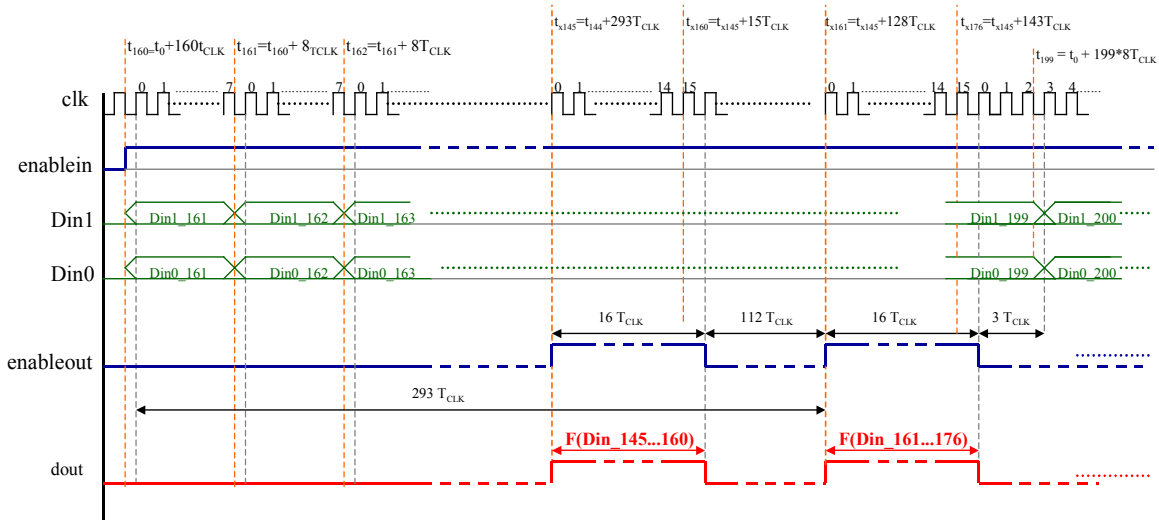


Figura 4.2: Interfaz del decodificador. decoderverilog.v.



Cronograma 4.1: decoderverilog.v, inicio de la trama. Decoding depth=32.



Cronograma 4.2: decoderverilog.v, final de la trama. Decoding depth=32.
Nota 4.2: La trama de entrada es de 200 símbolos.

Si la frecuencia de trabajo interna del decodificador es F_{CLK}, los datos deben llegar al decodificador con una frecuencia 8 veces menor. Por tanto entre dos datos consecutivos de entrada deben transcurrir 8 T_{CLKs}.

El primer dato de entrada al decodificador, x_1=(Din1_1,Din0_1), llega en el instante t_0 y el último de la trama de entrada, x_n=(Din1_n,Din0_n) en el instante $t_{n-1}=t_0+(n-1)*8T_{CLK}$.

enablein debe estar activo desde t_0 hasta t_n. No se puede desactivar enablein durante el transcurso de la trama de entrada, para pausar la decodificación está ce.

El bit decodificado correspondiente a x_1 estará disponible en la salida en el instante $t_{x1}= t_0 + 293\ T_{CLKs}$. Por tanto la latencia del decodificador es de 293 T_{CLK}.

Los bits decodificados se obtienen en la salida en bloques de 16 bits, separados cada uno 1 T_{CLK}.

El primer bloque corresponde a la decodificación de los símbolos de entrada x_1 a x_{16}. El bit decodificado correspondiente a x_1 se obtiene en $t_{x1} = t_0 + 293\ T_{CLK}$, t_0 es el instante en que llega x_1 a la entrada del decodificador.
Y el correspondiente a x_{15} en $t_{x15} = t_{x1} + 15\ T_{CLK}$.

El siguiente bloque corresponderá a la decodificación de los símbolos x_{17} a x_{32}. El bit decodificado correspondiente a x_{17} se obtiene en $t_{x17}=t_{16}+293\ T_{CLK}=t_{x1}+128T_{CLK}$, t_{16} es el instante en que llega x_{17} a la entrada del decodificador. Y el correspondiente a x_{32} en $t_{x32}= t_{x17} + 15\ T_{CLK}$.

El proceso continúa hasta el final de la trama de entrada. La decodificación se divide en bloques de 16 bits y hay una separación de 128 T_{CLK} entre cada bloque.

Si la trama de entrada es continua, no hay inconvenientes adicionales al utilizar este método. Pero si las tramas de entrada son finitas, esta técnica tiene un inconveniente añadido. Consiste en que los últimos bits de cada trama no se decodifican, porque sólo se pueden dividir los símbolos en bloques completos de 16 bits. Así que las tramas estarán divididas en dos partes:

1. Los símbolos con información que se decodifican.
2. A continuación, al final de la trama, los símbolos de relleno que no se decodifican.

Tabla 4.1: Bits de relleno al final de la trama de entrada.					
Nº símbolos de la trama de entrada = N= x*16 + y. N= nº símbolos decodificados + nº símbolos relleno.		Número de símbolos Decodificados		Símbolos de relleno	
Genérico	Ejemplo	Genérico	Ejemplo	Genérico	Ejemplo
Y =-11	197, x=13, y=-11	(x-2)*16	176, x=13	N-((x-2)*16)	21
Y =-10	198, x=13, y=-10				22
Y = -9	199, x=13, y=-9				23
Y= -8	200, x=13, y=-8				24
Y= -7	201, x=13, y=-7				25
Y= -6	202, x=13, y=-6				26
Y= -5	203, x=13, y=-5				27
Y= -4	204, x=13, y=-4				28
Y= -3	205, x=13, y=-3				29
Y= -2	206, x=13, y=-2				30
Y= -1	207, x=13, y=-1				31
Y= 0	208, x=13, y=0				32
Y= 1	209, x=13, y=1				33
Y= 2	210, x=13, y=2				34
Y= 3	211, x=13, y=3				35
Y= 4	212, x=13, y=4				36

En la tabla anterior observamos que el número de símbolos que no se decodifican no es fijo, sino que depende del tamaño de la trama de entrada. Entonces siempre debemos añadir unos símbolos de relleno al final de la trama.

Por ejemplo si la trama de entrada es de 200 símbolos, sólo se decodifican los primeros 176 símbolos, y los últimos 24 constituyen el relleno. 200 = (13-2)*16 + 24.

Para conseguir una decodificación independiente de la longitud de la trama, debemos añadir siempre 36 símbolos de relleno al final de la trama.

4.4.1 Definición puertos.

Tabla 4.2: Puertos decoderverilog.v		
Puerto	Dirección	Descripción
Clk	Entrada	Reloj del sistema. El período de proceso de un dato son 8 T_{CLKS}.
$\overline{aclr}$	Entrada	Reset asíncrono, activo a nivel bajo. Pone todos los puertos de salida, las señales internas y la memoria interna a cero. Si se activa mientras el decodificador está decodificando una trama, ya no se podrá reanudar la decodificación de la trama. Obligatorio activarlo durante al menos 8 T_{CLKs} antes del inicio de cada trama.
Ce	Entrada	Es el clock enable. Se utiliza para compatibilizar el diseño con System Generator. Si ce = '1', no afecta a la decodificación. Si ce = '0', entonces la decodificación se para, pero almacena su estado. De manera que cuando ce vuelva a ser uno, la decodificación se reanuda correctamente desde el estado almacenado. Es equivalente a una pausa y se puede activar y desactivar en cualquier momento incluso en medio de una trama.
enablein	Entrada	Indica si hay datos disponibles en la entrada, activo a nivel alto. Permanecerá activo siempre que haya datos disponibles en la entrada y desactivado en caso contrario. Debe estar activado o desactivado durante los 8 T_{CLKs} del período de proceso. No se puede desactivar enablein en medio de una trama de datos de entrada. Por tanto no se puede usar para pausar la decodificación. Porque con enablein desactivado, el decodificador interpreta que la trama de bits de entrada ha finalizado, y comienza a sacar los bits que tenga almacenados en la memoria del trellis.
din(1:0)	Entrada	Son los datos de entrada al decodificador. din(1) corresponde a dout1 del codificador convolucional, (polinomio 171, 1111001). din0 corresponde a dout0 del codificador convolucional, (polinomio 133, 1011011). Cada dato de entrada al decodificador, pareja din(1:0) debe mantenerse estable durante los 8 T_{CLKs} que dura un período de proceso.
enableout	Salida	Indica que el bit de la salida es válido, activo a nivel alto. Cuando está activo permanece a '1' durante un T_{CLK} y a '0' durante los restantes 7 T_{CLKs} de cada período de proceso.
dout	Salida	Bit de salida, que se corresponde a la decodificación de un dato compuesto por din(1:0).

4.5. Obtención decoderverilog.v.

4.5.1 Obtención del código a partir del generador automático Opencores.

Tras descargar todos los archivos de [1], debemos seguir estos pasos:

Poner en una carpeta las carpetas, C, data, module, perl y el archivo acs2.mod.
Ejecutar $>perl perl/morphsGUI.pl

Aparece una ventana en la que debemos poner estos parámetros:

- Polinomio: 121, 91. [121, 91 es el valor decimal de 177, 133 octal, (1111001 1011011 en binario)]. 64 + 32+ 16 +8 + 1= 121; 64 + 16 + 8 + 2+ 1= 91.

- V: Sólo se puede poner 1.

- B: De este valor dependen el área del decodificador y la velocidad. Si B aumenta, el área y la velocidad disminuyen. Y viceversa, si B disminuye, el área y la velocidad aumentan. El decodificador emplea 2^B T_{CLK} en decodificar 1 bit.

- OSR: La decodificación se organiza en bloques de salida de 2^{OSR} bits. Un valor normal es 3 ó 4. Valores de 1 y 2 no funcionan, los hemos probado y no se sintetizan bien.

- TBL es la profundidad de memoria, decoding depth. Tiene que ser múltiplo de 2^{OSR}.

- RAW el valor correcto es 10. Con un valor mayor el tiempo de síntesis aumenta muchísimo.

- Consultar specification sheet en [1] para ampliar la información sobre estos parámetros.

En nuestro caso hemos probado varias combinaciones y nos hemos decidido por
Polinomio(121,91); V(1); B(3); OSR(4); TBL(32); RAW (10)

El polinomio viene fijado por las especificaciones del decodificador, 171, 133 octal.
La especificación es decoding depth = 36. Pero no podemos conseguir ese valor exacto porque tiene que ser potencia de 2.

El resto de parámetros, V, B, OSR y RAW no vienen impuestos por las especificaciones del decodificador. Así que podemos modificarlos. Al final hemos elegido la combinación con la que mejores resultados hemos obtenido.

La herramienta genera automáticamente el código del decodificador con las especificaciones indicadas en los parámetros anteriores, y se organiza en las siguientes carpetas:

- En module se habrá generado el código del decodificador y un simulador en Verilog. El archivo principal del decodificador es *decoder0.v* y su test bench asociado es *testbench0.v*.

- En la carpeta c se genera un codificador convolucional: *encode8.cpp*. Al compilarlo se obtendrá *encode8.exe*.

- En data se generan tres ficheros, *code, source y f_filo_out*.
  - *code* es el fichero de entrada al *testbench0.v*. Contiene sus bits de entrada. El formato es 1 1 0 1 1 1:
    Din0(0) Din0(1) Din1(0) Din1(1) Din2(0) Din2(1)
    Van entrando los bits separados por un espacio.

  - *f_filo_out* es el fichero de salida de *testbench0.v*. Contiene sus bits de salida.

  - *source* son los bits de entrada al codificador.

  - *code* son los bits de entrada al decodificador. *f_filo_out* son los bits de salida del decodificador. Por tanto si todo funciona correctamente, *f_filo_out* debe ser igual a *source*.

Ejecutando $>encode8 <ficheroOrigen> ficheroDestino
En ficheroDestino (*code*) se obtienen los bits que haya en fichero origen (*source*) codificados con un codificador convolucional. FicheroOrigen debe tener el mismo formato que el fichero *souce* y fichero destino tendrá este formato:
1 0 1 1 1 0 0 …… Este es el formato adecuado para el fichero de entrada del *testbench0..v*

Para nosotros solamente tiene utilidad el propio código del decodificador, el *decoder0.v.* El resto de elementos que se generan automáticamente no nos sirven para el desarrollo del proyecto. El *testbench0.v* es un simulador muy básico que no utilizaremos. En su lugar desarrollaremos unos simuladores mucho más completos y eficientes: *TestSimulaciónCompleta.vhd* y *TestLeeFichero.vhd*.

El desarrollo de estos simuladores supone un valor añadido del proyecto puesto que aportan muchas ventajas y funcionalidades frente a un simulador básico como el *testbench0.v.*

4.5.2 Añadidos al código obtenido automáticamente.

Debemos realizar cambios en el código fuente original. Por tanto necesitamos conocer el lenguaje Verilog, cuyas referencias están en el *apartado 4.8.1*.

Desarrollamos nuestros propios simuladores *TestSimulaciónCompleta.vhd* y *TestLeeFichero.vhd*.

Desarrollamos un nuevo archivo en Verilog, el *decoderverilog.v*, que se convierte en el módulo principal del decodificador, ver *figura 4.2*. Gracias a este módulo modificamos los puertos de entrada y salida originales para ajustarlos exactamente a nuestras especificaciones.

Nuestra especificación es decodificación dura, 1 bit, con Din1 correspondiente al polinomio 171 y Din0 al 133. Sin embargo en *decoder0.v* se realiza una decodificación soft con 3 bits. Además symbol0 corresponde al polinomio 171 y symbol1 al 133. Hacemos los cambios necesarios mediante unos multiplexores, de manera que las entradas Din1 y Din0 de *decoderverilog.v* cumplen nuestras especificaciones.

Otros cambios que añadimos son:

- La entrada pattern la dejamos fija a "11".
- Cambiamos los nombres de algunos puertos para adaptarlos a la nomenclatura que utilizamos en el resto de módulos del proyecto. valid_in → enablein, valid_out → enableout, filo_out → dout y rst → aclr.

Cambios realizados en el código Verilog para que sea compatible con System Generator:

Todo código fuente HDL que se integre en System Generator debe cumplir unas condiciones. Estas condiciones están expuestas en la guía de referencia de System Generator, apartado black box, páginas 64 a 71 de la referencia [12] del *tema 1*.

Tras leer la documentación, sabemos que debemos realizar estos cambios:

- Hay que añadir un clock enable, ce, en todos los flip flops de cada uno de los módulos del código.
- Cambiar la denominación del reloj, mclk por clk en todos los módulos.
- Cambiar la indexación de todos los vectores. Inicialmente están indexados (0 to x), pero System Generator exige que sean (x downto 0).
- Quitamos el *glb_def.v* porque el System Generator no es capaz de encontrarlo al hacer la compilación. En su lugar ponemos las constantes que hagan falta en cada uno de los ficheros.
- Comentamos los ASSERTS, porque dan error.

4.6 Errores.

4.6.1 No se trata exactamente del algoritmo de Viterbi.

Este es el error más importante, la decodificación de los símbolos no se hace correctamente.

En los *apartados 2.5 y 2.6* vimos como es exactamente el algoritmo Viterbi. Básicamente consiste en:

1. En cada período de proceso llega un nuevo símbolo a la entrada del decodificador y se obtiene un bit decodificado en la salida.

2. Internamente los bits se almacenan formando una malla trellis con 64*(decoding depth) bits.

3. En la malla trellis hay 64 caminos supervivientes, uno hasta cada estado.

4. En cada período de proceso se introduce un bit en la última posición de cada camino superviviente y se desplazan una posición los que ya estaban. De manera que el que ocupaba la primera posición en cada uno de los caminos sale de la memoria.

5. Entre esos 64 bits que han salido de la memoria está el bit decodificado que saldrá en Data_Out del decodificador. Se corresponde al del camino que tenga una distancia menor en su última posición.

Pero en el decodificador de Opencores, el proceso es diferente. El fallo está en que la decodificación no se hace de manera continua, sino que se hace en bloques de 16 símbolos. El método es el siguiente:

1. En cada período de proceso llega un nuevo símbolo al decodificador, que se introduce en la malla trellis.

2. La forma en la que los símbolos se almacenan en la malla trellis es correcta, coincide con el algoritmo Viterbi exacto.

3. Cuando hayan llegado 32 símbolos, la malla estará completa.

4. Una vez completada la malla, se encuentra el camino superviviente con menor distancia en su última posición y se selecciona como el ganador.

5. Los primeros 16 bits del camino superviviente ganador constituyen los bits decodificados ganadores. Los 16 pasan a Data_Out inmediatamente, 1 bit en cada T_{CLK}. Este es el fallo del sistema, el Viterbi no debe hacer eso. Lo correcto sería decodificar solamente un bit en cada período de proceso, que correspondería al primer bit del camino superviviente ganador.

6. Los 16 primeros bits de cada uno de los 64 caminos salen de la malla trellis. De manera que esta queda con sólo 16 posiciones.

7. Durante los siguientes 16 períodos de proceso, se van almacenando los símbolos de entrada al decodificador. Llega un símbolo por período de proceso, y durante este tiempo no se obtiene ningún bit decodificado.

8. Cuando la malla vuelva a estar llena con 32 posiciones ocupadas, se pasa al punto 4. Y los pasos se van repitiendo indefinidamente.

4.6.2 Demostración del error utilizando un ejemplo con 4 estados.

Utilizamos un sistema de comunicaciones descrito en el *apartado 2.3.1*. Las características del codificador convolucional son:

- Constraint length K=7. Definida como la longitud en bits del polinomio generador, el codificador tiene 2 registros de memoria.
- Code rate R = 1/2 (tasa = 1/2). 1 bit de entrada, 2 de salida.
- 4 estados.
- R=1/2; 1 bit de entrada m[n] y 2 de salida $Dout_1$ y $Dout_0$.
- D_{out1}= 101 binario. $Dout_1$ = m[n] + m[n-2].
- D_{out0}= 111 binario. $Dout_0$ = m[n] + m[n-1] + m[n-2]

Secuencia original de entrada al codificador:
m[n] = {1,1,0,0,1,0,1,0....} = {m[0], m[1], m[2], m[3], m[4], m[5], m[6], m[7],...}, siendo m[0] el primer bit en entrar al codificador.

Secuencia codificada original, a la salida del codificador:
$Dout_1[n]Dout_0[n]$ = {11, 10, 10, 11, 11, 01, 00, 01,....}

Secuencia que llega al decodificador, tras añadirle el ruido.
$Din_{Decoder1}[n+t_x]Din_{Decoder0}[n+t_x]$ = x ={11, 10, 00, 10, 11, 01, 00, 00,...}
= {x_1,x_2,x_3,x_4,x_5,x_6,x_7,x_8,....}; en rojo los bits que se reciben con errores.

t_x es el tiempo que transcurre desde que un bit está disponible en la entrada del codificador; hasta que el símbolo correspondiente a la codificación de ese bit llega a la entrada del decodificador. A cada símbolo de entrada del decodificador lo llamamos x.

Tabla 4.3: Ejemplo 4 estados, bits en la entrada del decodificador.

Instante en el que llegan los Símbolos al decodificador.	$n=t_x$ t_0	$n=t_{x+1}$ $t_1=t_0+8T_{CLKs}$	$n=t_{x+2}$ t_2	$n=t_{x+3}$ t_3	$n=t_{x+4}$ t_4	$n=t_{x+5}$ t_5	$n=t_{x+6}$ t_6	$n=t_{x+7}$ t_7
$Din_{Decoder1}[n]Din_{Decoder0}[n]=x$	x_1=11	x_2=10	x_3=00	x_4=10	x_5=11	x_6=01	x_7=00	x_8=00

El objetivo del decodificador es obtener en su salida una cadena de bits igual a la de entrada al codificador. DataOutViterbi[$n+t_y$]= m[n] = F(x[n])
Siendo $t_y=t_x$ + latencia decodificador.

Tanto el decodificador UC3M como el de Opencores almacenan correctamente los símbolos recibidos en la malla trellis, por lo que en los dos casos tendremos este resultado:

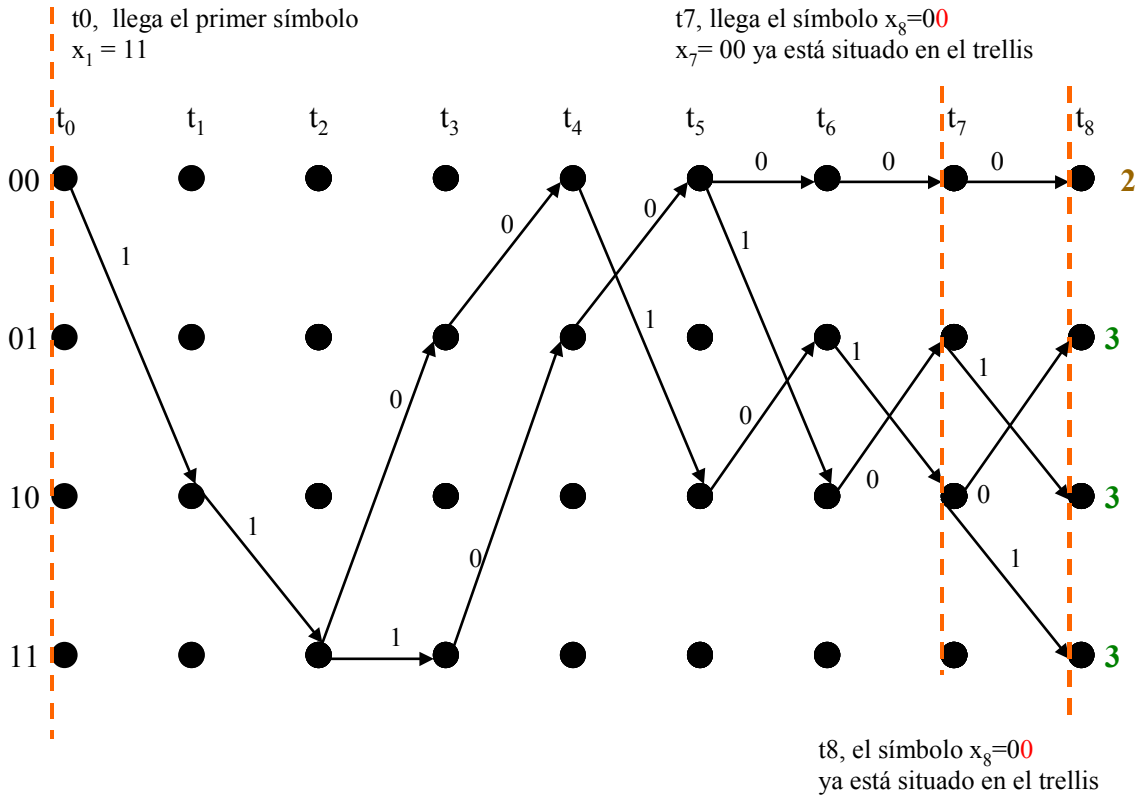


Figura 4.3: Malla trellis completa, ejemplo 4 estados.

La diferencia en el funcionamiento está en el modo en el que el decodificador examina la malla trellis para obtener los bits decodificados:

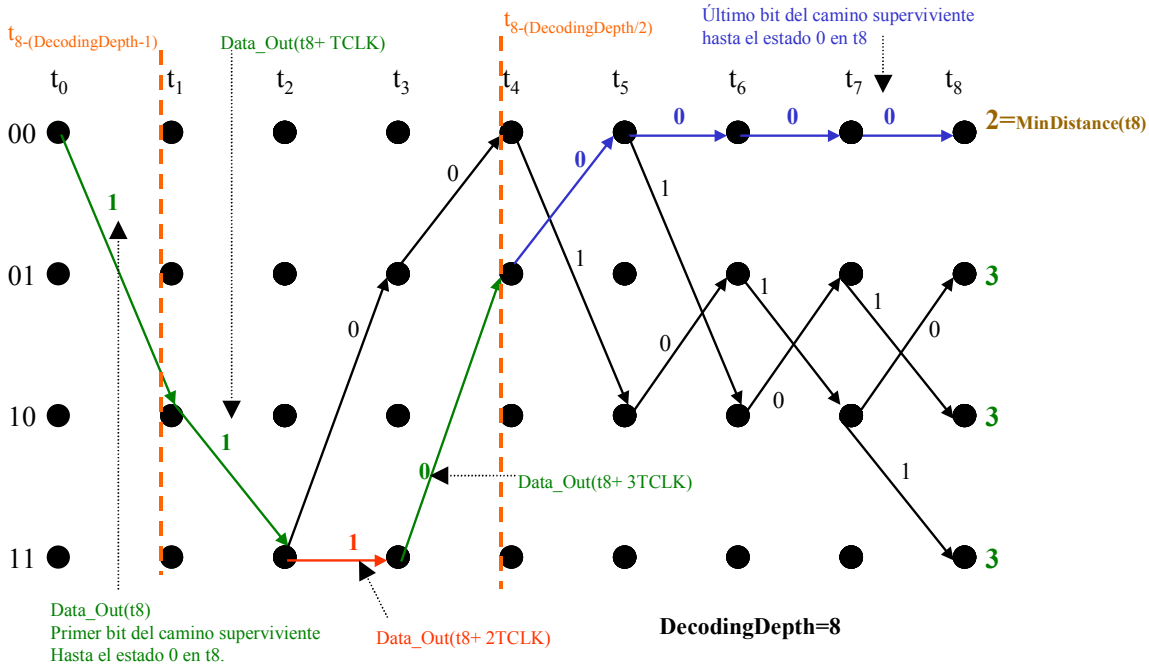


Figura 4.4: Extracción de los bits decodificados en el modelo de Opencores.

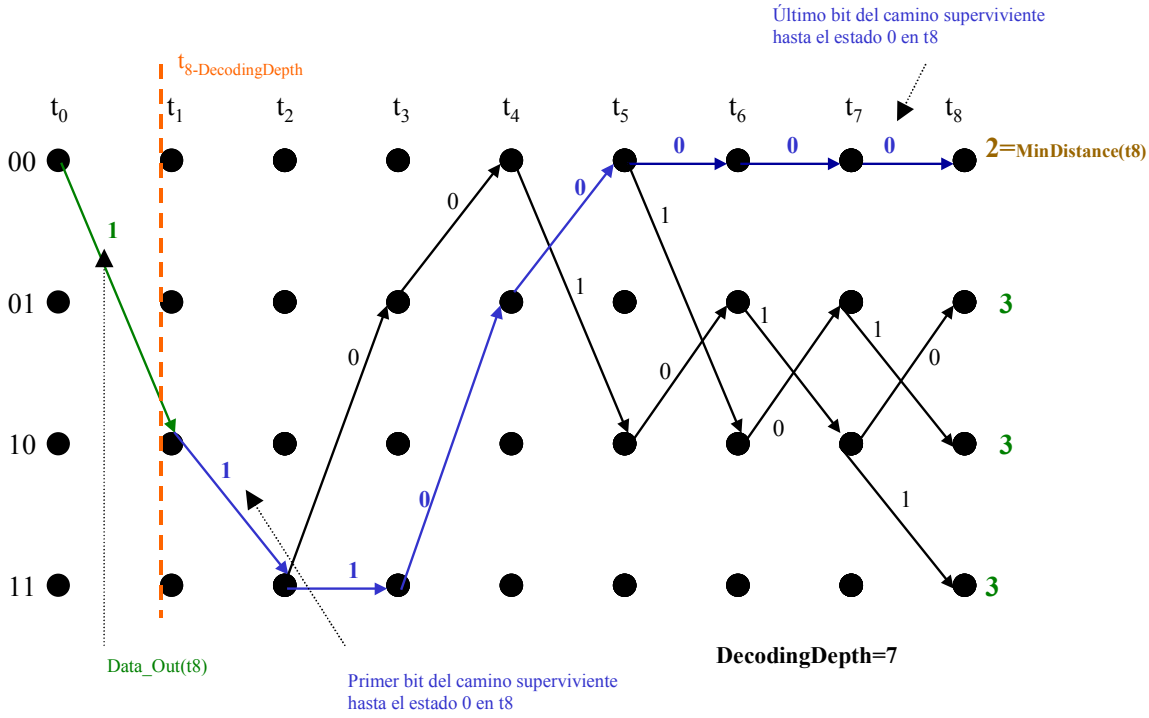


Figura 4.5: Extracción del bit decodificado en un algoritmo Viterbi exacto.

La secuencia original de entrada al codificador es:
m[n] = {1,1,0,0,1,0,1,0....}

El decodificador Viterbi exacto ha obtenido Data_Out(t_8)= '1'. Es correcto porque es el primer bit decodificado y corresponde con el primer bit en la entrada al codificador → Data_Out[t8]=m[0] = '1' → Correcto . El resto de bits se irán obteniendo a continuación, un bit decodificado en la salida por cada período de proceso.

Sin embargo el decodificador de Opencores obtiene 4 bits al mismo tiempo, presentándose en la salida del decodificador con una separación de 1 T_{CLK}.
Data_Out_Opencores={1,1,1,0}={Data_Out(t_8),Data_Out(t_8+T_{CLK}),
Data_Out($t_{8+}2T_{CLK}$), Data_Out($t_{8+}3T_{CLK}$)}

Data_Out_Opencores(t_8) = m[0] = '1' → Correcto.
Data_Out_Opencores(t8+T_{CLK}) = m[1] = '1' → Correcto.
Data_Out_Opencores($t_8 + 2T_{CLK}$) = '1' ≠ m[2] = '0' → Fallo.
Data_Out_Opencores (t_8 + 3 T_{CLK}) = m[3] = '0' → Correcto.

El error se debe a que en la malla trellis los primeros bits de los caminos convergen en una misma rama, en la figura el instante entre t_0 y t_2. Pero a continuación los caminos divergen hasta llegar al último bit del camino, que representa el bit situado en el trellis en el momento actual, t_x. Cuanto más alejado de t_x esté el bit decodificado mejor, porque así hay más posibilidades de que los caminos converjan hasta el valor adecuado, y así la decodificación se realice sin errores.

En un algoritmo Viterbi, el bit decodificado en el instante t_x está en la posición $t_{x\text{-DecodingDepth}}$.

En el de Opencores se extraen bits desde $t_{x\text{-(DecodingDepth/2)}}$ hasta $t_{x\text{-(DecodingDeth-1)}}$. Entonces no se están utilizando las ventajas del Viterbi porque no se está utilizando una malla trellis con decoding depth posiciones, sino que únicamente tiene DecodingDepth/2 posiciones.

4.6.3 Hay latches.

Al sintetizar se obtienen 16 latches de 16 bits y 2 de 3 bits. Entonces el diseño no es el adecuado y no se cumplen las reglas de diseño síncrono. Esto implica que no tenemos garantía de que el comportamiento real en la placa sea igual al que obtenemos en la simulación. Por tanto se trata de un fallo importante, y las probabilidades de que el circuito falle al pasarlo a la placa son mucho mayores que en un diseño sin latches. Referencias sobre diseño síncrono y latches en el *tema 1*, citas [63], [64], [65] y [66].

No nos preocupamos de eliminar los latches debido a que nos llevaría mucho tiempo y no compensa el esfuerzo, puesto que sólo utilizaremos *decoderverilog.v* de manera temporal. Además ya hemos visto en el apartado anterior que su rendimiento es muy bajo, así que aún corrigiendo los latches, el *decoderverilog* seguiría sin resultarnos útil.

4.7 Comparación con un decodificador Viterbi exacto.

En el punto anterior hemos visto que el decodificador de Opencores no es exactamente un algoritmo de Viterbi, y sabemos que su capacidad de corrección de errores será menor que la del auténtico algoritmo.

Una vez implementado, desarrollamos un simulador que nos permite comparar las características de Opencores con otro decodificador que tomamos como referencia, el IP core Viterbi v5.0 de Xilinx. Este simulador es una parte muy importante del proyecto, y lo explicaremos detalladamente en el *tema 6*. En este momento únicamente mostramos un avance de las medidas en la siguiente figura. El conjunto completo de medidas está detallado en el *capítulo 7*.

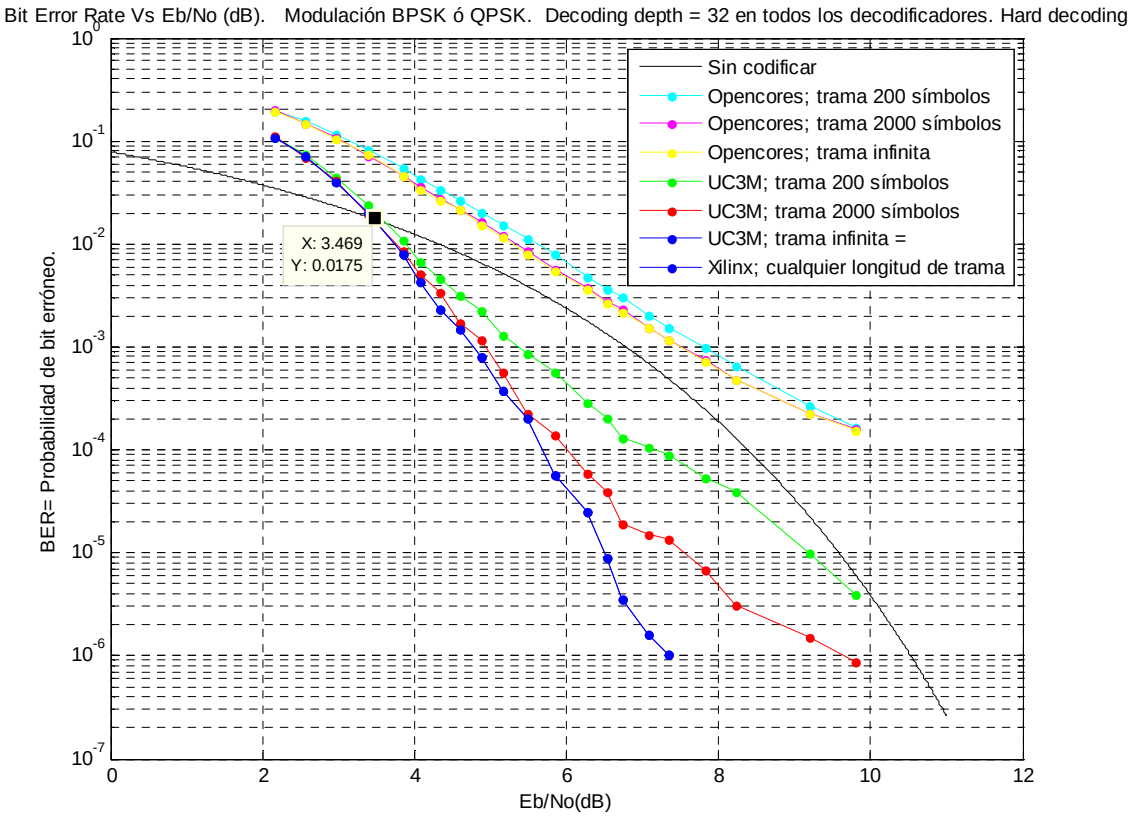


Figura 4.6: BER(Eb/No). Decodificadores Opencores y UC3M, decoding depth=32.

Analizando las medidas observamos:

- Si al decodificador llegan bits de manera continua, el UC3M se comporta exactamente igual que el de Xilinx. Esto es lo que debe pasar porque ambos implementan el mismo algoritmo, así que los resultados deben ser los mismos. Por eso sus representaciones utilizan la misma línea.

- Sin embargo en el de Opencores la capacidad de corrección de errores es mucho menor. Esto lo esperábamos, aunque hemos de reconocer que nos ha sorprendido un rendimiento tan sumamente bajo. Su uso no mejora la BER frente a un sistema sin codificar, por lo que empeora las prestaciones del sistema. Por tanto a la vista de la *figura 4.6*, el uso del Viterbi de Opencores no tiene ningún sentido, puesto que aumentamos la complejidad del sistema y empeoramos las prestaciones. En el *apartado 7.7.5* veremos que este decodificador debe utilizarse siempre con decodificación soft. Esta es la única forma de que la BER mejore frente al sistema sin codificar.

- La capacidad de corrección de errores del decodificador de Xilinx es independiente del tamaño de la trama de entrada, mientras que en los otros 2 no es así.

- Si la cadena de entrada se divide en tramas, entonces el *ViterbiDecoder.vhd* no corrige tantos errores como el de Xilinx. Esto no es un error, lo que ocurre es que el *ViterbiDecoder.vhd* emplea la técnica de truncamiento de trellis, mientras que el de Xilinx emplea cola de ceros. Los dos métodos son igualmente válidos, y cada uno tiene sus ventajas e inconvenientes, en el *apartado 7.7.4* explicamos la diferencia entre las dos técnicas. También se explica en las referencias [40] y [41] del *tema 7*.

- El *decoderverilog.v* tambien se emplea truncamiento de trellis y por eso el rendimiento empeora cuando el tamaño de la trama de entrada disminuye.

4.8 Referencias.

Proporcionamos el enlace Web siempre que exista, accedemos a estos enlaces por última vez en Febrero de 2011. Además tenemos una copia de seguridad en el CD del proyecto de toda la documentación sin copyright.

[1] Especificación decodificador Viterbi Opencores:
Mikael Johnson. "Specification of Viterbi HDL Code Generator". 27 Mayo de 2004.
La documentación y el código del decodificador Viterbi de Opencores están accesibles en la Web de Opencores. El decodificador Viterbi está en la categoría arithmetic core.

Sitio Web Opencores: http://opencores.org/

Sitio Web Viterbi Opencores. http://opencores.org/project,vhcg

Además tenemos una copia de seguridad de la specification sheet en el CD del proyecto.

4.8.1 Verilog.

[2] Norma IEEE Std 1364–2001. "IEEE Standard Verilog Hardware Description Languaje". Aprobado el 28 de Septiembre de 2001 .
http://www.sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf

http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8238

[3] "Verilog-A Language Reference Manual. Analog Extensions to Verilog HDL". Puplicado por Open Verilog International. Version 1.0, 1 Agosto 1996.

[4] "The Verilog Golden Reference Guide". Doulos, 1996.

[5] Rajeev Madhavan. "Verilog Reference Manual".Automata Publishing Company. Última actualización 14 Abril 1997.
http://eesun.free.fr/DOC/VERILOG/verilog_manual1.html

[6] Stuart Sutherland. "Online Verilog-1995 Quick Reference Guide". Sutherland HDL, Inc, 1995.
http://www.sutherland-hdl.com/online_verilog_ref_guide/vlog_ref_top.html

http://sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf

http://onesutherland.com/reference-guides.php

[7] Rajeev Madhavan"Quick Reference for Verilog HDL". AMBIT Design Systems, Inc, 1995.
http://www.stanford.edu/class/ee183/handouts_win2003/VerilogQuickRef.pdf

[8] Página Web con enlaces a cursos, tutoriales y referencias. Copyright 2011 Verilog.com.
http://www.verilog.com/
http://www.verilog.com/IEEEVerilog.html
http://vol.verilog.com/

[9] Stuart Sutherland. "The IEEE Verilog 1364-2001 Standard What's New, and Why You Need It" Sutherland HDL, Inc, October 2001.

[9A]http://www.sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf

[9B] Versión en diapositivas:
http://ce.sharif.ir/courses/84-85/1/ce223/resources/root/Slides/11-Verilog-2001.pdf

[10] "FPGA Compiler II/ FPGA Express. Verilog HDL Reference Manual". Synopsys, version 1995.05, May 1999.

[11] Douglas J. Smith. "A Practical Guide for Designing, Synthesizing & Simulating Asics & FPGAs Using Vhdl or Verilog". Donne Publications, Marzo 1998.

CAPÍTULO 5

5. IMPLEMENTACIÓN DECODIFICADOR UC3M: ViterbiDecoder.vhd

5.1 Características.

Parámetros importantes:

- Constraint length K=7. Definida como la longitud en bits del polinomio generador. El codificador tiene (K-1) = 6 registros de memoria. 2^{K-1}=64 estados.
- Decode rate R = ½, (tasa = 1/2). 2 bits de entrada, 1 de salida.
- Polinomio generador 171, 133 (octal). 1111001 y 1011011.
- Decoding depth (Profundidad de memoria). Número de etapas que el decodificador guarda en la memoria antes de decodificar un bit. Debe ser mayor que K-1 = 6. Para modificar este parámetro únicamente hay que cambiar el valor de dos constantes en *Constantes.vhd*. El decodificador de nuestras especificaciones es de 36. Pero al ser tan fácil de modificar hemos realizado también decodificadores con decoding depth = 8, 24, 32, 48 y 60. Hemos sintetizado y simulado todos, resultados en *apartado 7.7.3*.
- Decodificación dura.
- Período de proceso de un dato en el decodificador = 8 T_{CLKs}. Es el número de T_{CLKs} que necesita el decodificador entre dos datos de entrada consecutivos.

  Este período de proceso indica que el decodificador debe trabajar internamente con una frecuencia 8 veces mayor que la frecuencia con la que llegan datos a su entrada. Por tanto en la entrada habrá un dato nuevo cada 8 T_{CLKs} y en la salida habrá un bit decodificado cada 8 T_{CLKs}. Pero internamente el decodificador trabaja con período T_{CLK}.
- En la entrada el dato debe estar estable durante 8 T_{CLKs}. En la salida por cada dato se genera un pulso de 1 T_{CLK}, y los 7 restantes estarán a cero.

- Latencia 10 + (decoding depth+1)*8 T_{CLKs}
  - Si decoding depth = 36 → 306 T_{CLK}. (306/8)=38,25 períodos de proceso.
  - Si decoding depth = 60 → 498 T_{CLKs}. (82/8)=62,25 períodos de proceso.

- Los T_{CLKs} de latencia se reparten de la siguiente manera:
  - 1 por las entradas registradas en el módulo *ViterbiDecoder.vhd*.
  - 8 del *ACS.vhd*.
  - 1 +((decoding depth+1)*8) del *RegisterExchange.vhd.*

- La latencia es el número de TCLKs que emplea el decodificador en decodificar un dato. Su valor se mantiene constante para todos los datos.
- Puede decodificar una cadena de símbolos continua o dividida en tramas. Utiliza la técnica truncamiento de trellis, esto significa que en la trama de entrada debe haber únicamente símbolos de datos. No hay que añadir símbolos de relleno ni al inicio ni al final de la trama.
- Tamaño mínimo de la trama de entrada ≥ decoding depth + 1.
- Entradas y salidas registradas.
- Estructura serie y paralelo. Consta de 8 etapas serie y en cada una de ellas se trabaja con 8 estados en paralelo. Con este sistema conseguimos el mejor compromiso entre área y velocidad. El sistema es más rápido cuantos más estados pueda decodificar en paralelo, pero también aumenta el área.
- Entre una etapa serie y la siguiente, hay registros de pipeline. Por ese motivo, son necesarios 8 T_{CLKs} entre dos datos de entrada consecutivos.

5.2 Interfaz Entrada/Salida. Uso del decodificador.

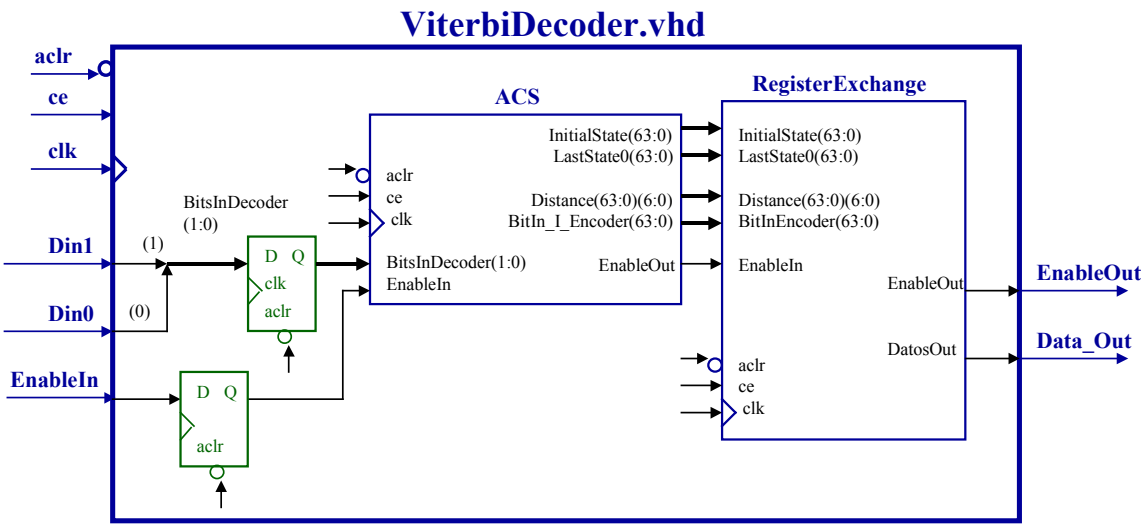
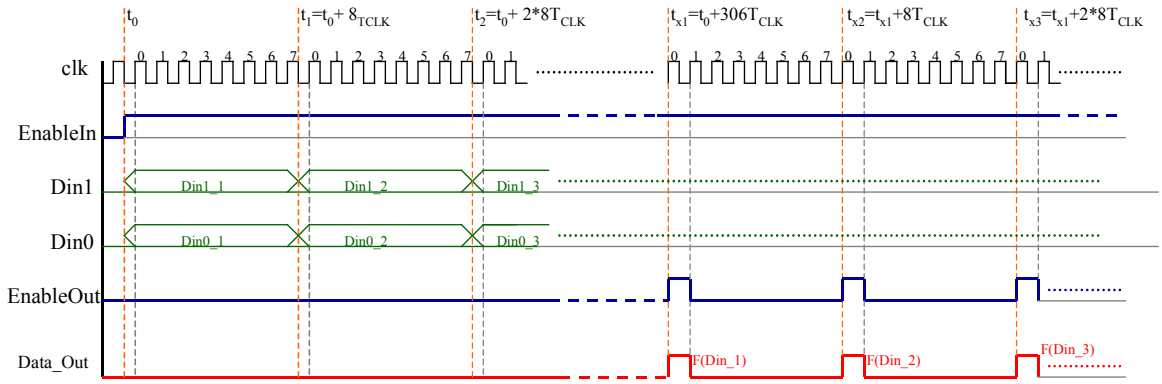
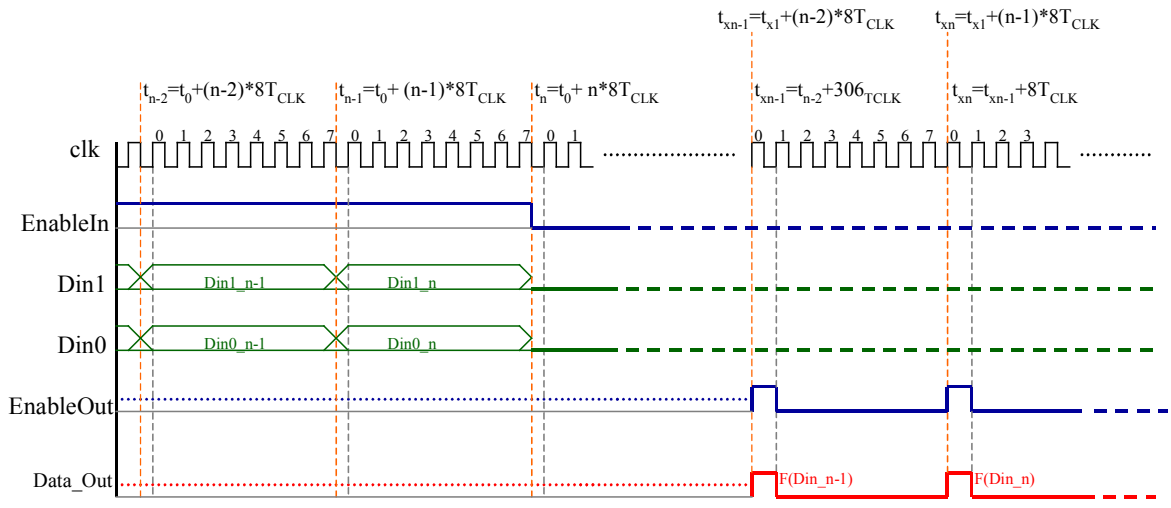


Figura 5.1: Interfaz del decodificador. ViterbiDecoder.vhd.



Cronograma 5.1: ViterbiDecoder.vhd, inicio de la trama. Decoding depth=36.



Cronograma 5.2: ViterbiDecoder.vhd, final de la trama. Decoding depth=36.

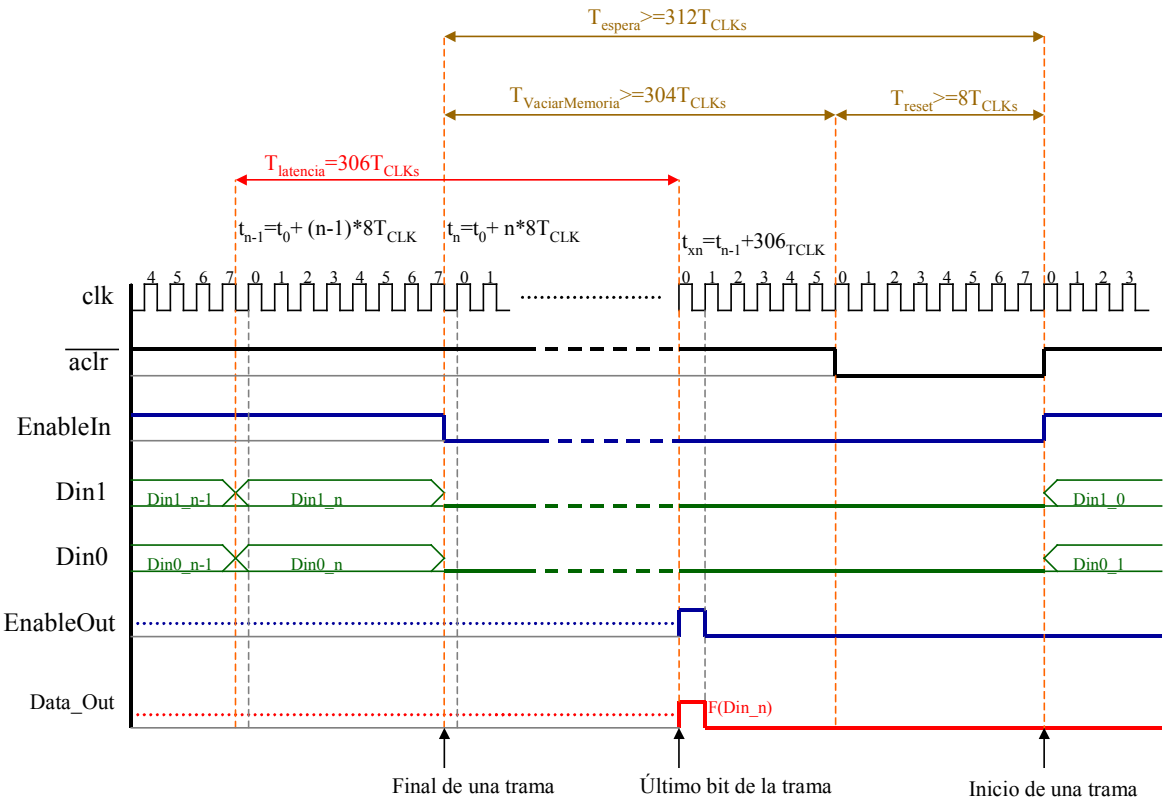
Si la frecuencia de trabajo interna del decodificador es F_{CLK}, los datos deben llegar al decodificador con una frecuencia 8 veces menor. Por tanto entre dos datos consecutivos de entrada deben transcurrir 8 T_{CLKs}.

El primer dato de entrada al decodificador, x_1=(Din1_1,Din0_1), llega en el instante t_0 y el último de la trama de entrada, x_n=(Din1_n,Din0_n) en el instante $t_{n-1}=t_0+(n-1)*8T_{CLK}$.

EnableIn debe estar activo desde t_0 hasta t_n. No se puede desactivar EnableIn durante el transcurso de la trama de entrada, para pausar la simulación está ce.

El bit decodificado correspondiente a x_1 estará disponible en la salida en el instante t_{x1}, y el bit correspondiente a la decodificación del dato x_n, en el instante:
$t_{xn}=t_{x1} + (n-1)*8T_{CLK}$.

$$\{t_{x1} = (10 + (DecodingDepth + 1) * 8)T_{CLK}\} \Leftrightarrow \begin{cases} 306T_{CLK} \rightarrow \text{Si DecodingDepth} = 36 \\ 82T_{CLK} \rightarrow \text{Si DecodingDepth} = 8 \end{cases}$$



Cronograma 5.3: ViterbiDecoder.vhd, intervalo entre tramas. Decoding depth=36.

En caso de dividir la decodificación en tramas, entre 2 tramas consecutivas debe haber una distancia mínima: T_{espera}≥ decoding depth + 3 períodos de proceso de un dato, 312 T_{CLKs} si decoding depth = 36.

$T_{espera} = T_{VaciarMemoria} + T_{reset}$.

- TvaciarMemoria ≥ decoding depth + 2 períodos de proceso de un dato, $304T_{CLKs}$ si decoding depth = 36.
  Durante todo este tiempo EnableIn y $\overline{aclr}$ están desactivados.
  Esta espera mínima obligatoria se debe a que cuando finalizan los datos de entrada de una trama, el decodificador sigue trabajando hasta que termine de decodificar los datos que tenga almacenados en su memoria.

- $T_{reset} \geq 8\ T_{CLKs}$.
  Durante este tiempo $\overline{aclr}$ está activado y EnableIn desactivado.
  Es obligatorio activar el reset para que al iniciarse la siguiente trama, el decodificador sitúe el primer dato de la trama en la posición inicial de la malla trellis.

Al iniciarse la siguiente trama, se activa EnableIn y se desactiva $\overline{aclr}$.

5.2.1 Definición puertos.

Tabla 5.1: Puertos ViterbiDecoder.vhd

Puerto	Dirección	Descripción
clk	Entrada	Reloj del sistema. El período de proceso de un dato son 8 T_{CLKS}.
$\overline{aclr}$	Entrada	Reset asíncrono, activo a nivel bajo. Pone todos los puertos de salida, las señales internas y la memoria interna a cero. Si se activa mientras el decodificador está decodificando una trama, ya no se podrá reanudar la decodificación de la trama. Obligatorio activarlo durante al menos 8 T_{CLKs} antes del inicio de cada trama.
ce	Entrada	Es el clock enable. Se utiliza para compatibilizar el diseño con System Generator. Si ce = '1', no afecta a la decodificación. Si ce = '0', entonces la decodificación se para, pero almacena su estado. De manera que cuando ce vuelva a ser uno, la decodificación se reanuda correctamente desde el estado almacenado. Es equivalente a una pausa y se puede activar y desactivar en cualquier momento incluso en medio de una trama.
EnableIn	Entrada	Indica si hay datos disponibles en la entrada, activo a nivel alto. Permanecerá activo siempre que haya datos disponibles en la entrada y desactivado en caso contrario. Debe estar activado o desactivado durante los 8 T_{CLKs} del período de proceso. No se puede desactivar EnableIn en medio de una trama de datos de entrada. Por tanto no se puede usar para pausar la decodificación. Porque con EnableIn desactivado, el decodificador interpreta que la trama de símbolos de entrada ha finalizado, y comienza a sacar los bits que tenga almacenados en la memoria del trellis.

Din1 y Din0	Entrada	Son los datos de entrada al decodificador. Din1 corresponde a Dout1 del codificador convolucional, (polinomio 171, 1111001). Din0 corresponde a Dout0 del codificador convolucional, (polinomio 133, 1011011). Cada dato de entrada al decodificador, pareja Din1 y Din0 debe mantenerse estable durante los 8 T_{CLKs} que dura un período de proceso.
EnableOut	Salida	Indica que el bit de la salida es válido, activo a nivel alto. Cuando está activo permanece a '1' durante un T_{CLK} y a '0' durante los restantes 7 T_{CLKs} de cada período de proceso.
Data_Out	Salida	Bit de salida, que se corresponde a la decodificación de un dato compuesto por Din1 y Din0.

5.3 Codificación del estado i y de sus anteriores j y h.

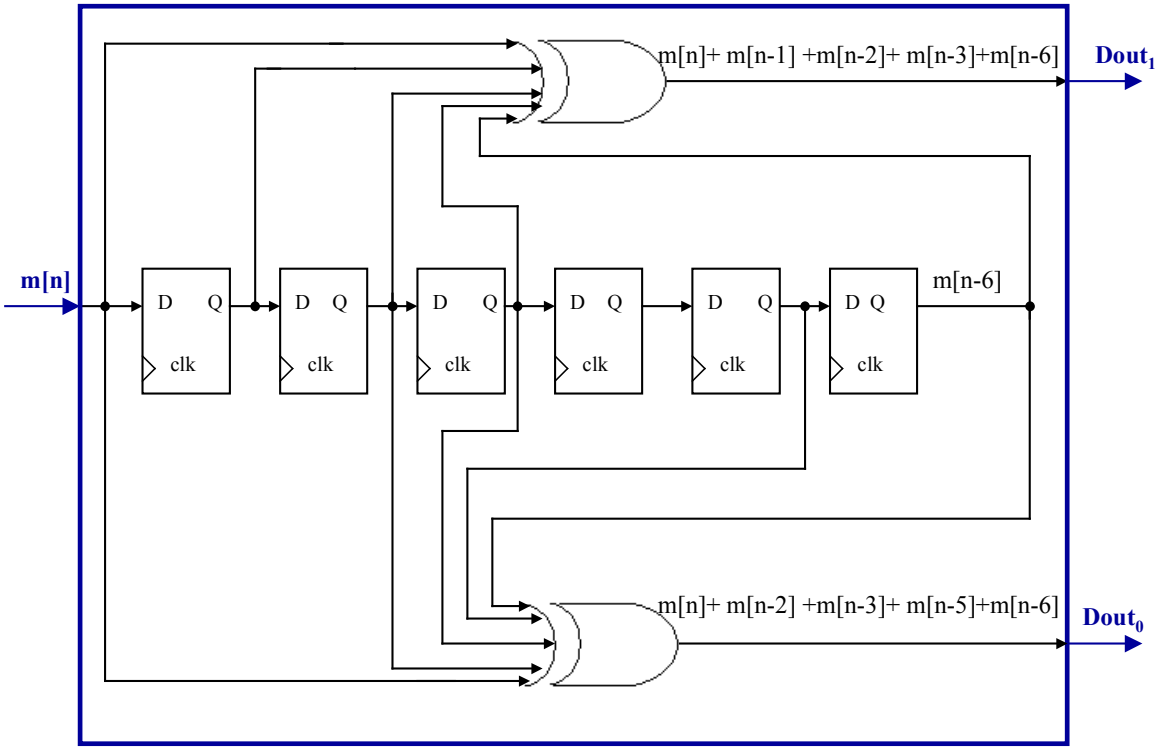


Figura 5.2: Codificador convolucional. K=7.

- El codificador es una máquina de estados, tiene 64 estados, 2^{K-1}.
- Un estado se codifica con 6 bits, m[n-1] a m[n-6].

- Se cumple que todo estado i tiene 2 estados posteriores, l y m. De manera que si en el instante t_x estamos en el estado i, un período de proceso después en el instante t_{x+1}, estaremos en el estado l o en el m. Dependiendo de si la entrada m[n] es cero o es uno.

- Al aplicar a un estado i en t_x una entrada m[n], tendremos dos salidas en el codificador en t_{x+1}, $Dout_1$ y $Dout_0$.

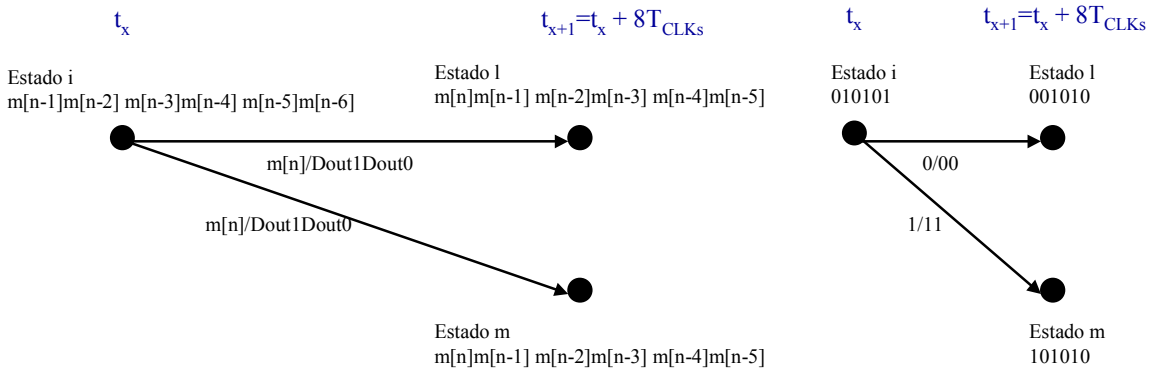


Figura 5.3 : Representación de los estados posteriores a i.

El cálculo de los estados posteriores a i es la base del codificador convolucional. Sin embargo para el decodificador no nos interesa la transición de un estado a su estado posterior. Porque para realizar la decodificación, debemos ir hacia atrás en el tiempo en la malla trellis. Explicación *en apartados 2.6 y 2.7*. Entonces en el decodificador lo que nos interesa es que cada estado i tiene 2 estados anteriores, j y h.

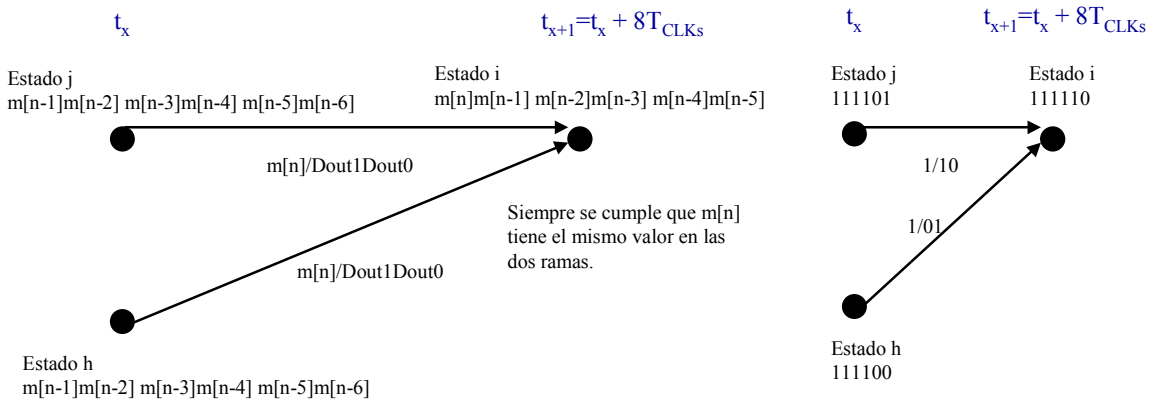


Figura 5.4 : Representación de los estados anteriores a i.

Todo este proceso está explicado en el *apartado 2.6*. Se trata de la base de la codificación convolucional, y puede ampliarse información en la bibliografía que citamos en el *apartado 5.6*: [1], [2], [3] y [4]. Además al tratarse de un aspecto básico de la codificación convolucional, también puede consultarse en cualquier referencia del *apartado 2.8.1*.

La estructura de datos básica del decodificador es la constante MatrixLastStates. Se trata de una matriz con 64 filas*2 columnas*9bits en cada columna. En cada fila i de la matriz almacenamos toda la información relativa a ese estado i y a sus dos anteriores, j y h:

- Columa 1:
  1. MatrixLastStates(i)(1)(8 downto 3): Codificación del estado j, de manera que 8 corresponde al valor m[n-1] y 3 al valor m[n-6].
  2. MatrixLastStates(i)(1)(2): El bit de entrada al codificador m[n], que permite pasar del estado j al i.
  3. MatrixLastStates(i)(1)(1): La salida Dout1, al aplicar la entrada m[n] al estado j. Polinomio 171.
  4. MatrixLastStates(i)(1)(0): La salida Dout0, al aplicar la entrada m[n] al estado j. Polinomio 133.

- Columa 0:
  1. MatrixLastStates(i)(0)(8 downto 3): Codificación del estado h, de manera que 8 corresponde al valor m[n-1] y 3 al valor m[n-6].
  2. MatrixLastStates(i)(0)(2): El bit de entrada al codificador m[n], que permite pasar del estado h al i.
  3. MatrixLastStates(i)(0)(1): La salida Dout1, al aplicar la entrada m[n] al estado h. Polinomio 171.
  4. MatrixLastStates(i)(0)(0): La salida Dout0, al aplicar la entrada m[n] al estado h. Polinomio 133.

Ejemplo en la fila i=62 tendremos MatrixLastStates(62) = (111101110 111100101).

La explicación es que el estado i=62=”111110” tiene como anteriores:

- j = ”111101” = MatrixLastStates(62)(1)(8 downto 3).
- h = ”111100” = MatrixLastStates(62)(0)(8 downto 3).

Al aplicar al estado j la entrada m[n]='1' = MatrixLastStates(62)(1)(2), el siguiente estado es i=”111110” y las salidas son:

1. Dout1 = '1' = MatrixLastStates(62)(1)(1).
2. Dout0 = '0' = MatrixLastStates(62)(1)(0).

Al aplicar al estado h la entrada m[n]='1' = MatrixLastStates(62)(0)(2), el siguiente estado es i=”111110” y las salidas son:

1. Dout1 = '0' = MatrixLastStates(62)(0)(1).
2. Dout0 = '1' = MatrixLastStates(62)(0)(0).

Tabla 5.2: MatrixLastStates					
Estado anterior j t_x	Estado anterior h t_x	Estado i, t_{x+1}	Estado anterior j t_x	Estado anterior h t_x	Estado i, t_{x+1}
111111111	111110100	111111; 63	111111000	111110011	011111; 31
111101110	111100101	111110; 62	111101001	111100010	011110; 30
111011111	111010100	111101; 61	111011000	111010011	011101; 29
111001110	111000101	111100; 60	111001001	111000010	011100; 28
110111100	110110111	111011; 59	110111011	110110000	011011; 27
110101101	110100110	111010; 58	110101010	110100001	011010; 26
110011100	110010111	111001; 57	110011011	110010000	011001; 25
110001101	110000110	111000; 56	110001010	110000001	011000; 24
101111100	101110111	110111; 55	101111011	101110000	010111; 23
101101101	101100110	110110; 54	101101010	101100001	010110; 22
101011100	101010111	110101; 53	101011011	101010000	010101; 21
101001101	101000110	110100; 52	101001010	101000001	010100; 20
100111111	100110100	110011; 51	100111000	100110011	010011; 19
100101110	100100101	110010; 50	100101001	100100010	010010; 18
100011111	100010100	110001; 49	100011000	100010011	010001; 17
100001110	100000101	110000; 48	100001001	100000010	010000; 16
011111101	011110110	101111; 47	011111010	011110001	001111; 15
011101100	011100111	101110; 46	011101011	011100000	001110; 14
011011101	011010110	101101; 45	011011010	011010001	001101; 13
011001100	011000111	101100; 44	011001011	011000000	001100; 12
010111110	010110101	101011; 43	010111001	010110010	001011; 11
010101111	010100100	101010; 42	010101000	010100011	001010; 10
010011110	010010101	101001; 41	010011001	010010010	001001; 9
010001111	010000100	101000; 40	010001000	010000011	001000; 8
001111110	001110101	100111; 39	001111001	001110010	000111; 7
001101111	001100100	100110; 38	001101000	001100011	000110; 6
001011110	001010101	100101; 37	001011001	001010010	000101; 5
001001111	001000100	100100; 36	001001000	001000011	000100; 4
000111101	000110110	100011; 35	000111010	000110001	000011; 3
000101100	000100111	100010; 34	000101011	000100000	000010; 2
000011101	000010110	100001; 33	000011010	000010001	000001; 1
000001100	000000111	100000; 32	000001011	000000000	000000; 0

Todos los valores de la tabla los escribimos a mano en la constante MatrixLastStates.

Los valores pueden extraerse de [5], pero no lo hacemos así porque se trata de un proceso lento y propenso a errores. Por este motivo desarrollamos unos módulos en VHDL, *ver anexo A.7*, que calculan automáticamente todos los valores de la tabla. Estos módulos no se incluyen en el código del decodificador, *ViterbiDecoder.vhd,* porque no son necesarios. Este sistema presenta varias ventajas:

1. Simplificamos la arquitectura del decodificador disminuyendo el área.

2. Como el decodificador no debe calcular ninguno de los valores, aumenta la velocidad del diseño y se disminuye la latencia.

3. Hemos desarrollado unas herramientas con Matlab para asegurarnos de que los valores calculados son correctos. Este proceso de validación es mucho más sencillo al hacerse el cálculo en un módulo independiente del decodificador.

4. Al ser una constante, el desarrollo del código es más simple, porque esta información debe ser accesible por casi todos los bloques del decodificador. Y como es una constante, la visibilidad es inmediata, no es necesario introducir los valores por un puerto de entrada.

5.4 Arquitectura y funcionamiento del sistema.

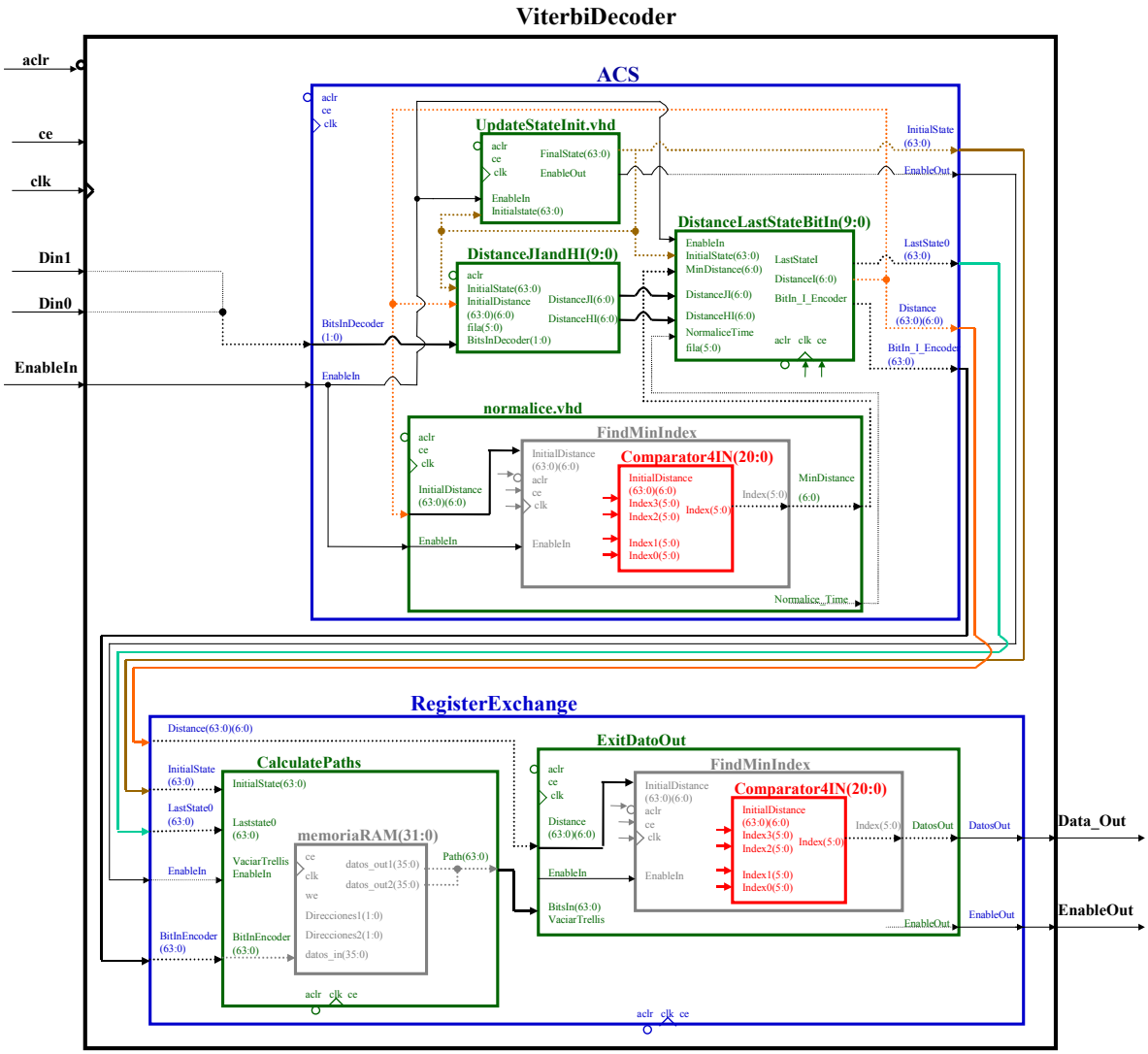


Figura 5.5: Arquitectura ViterbiDecoder.vhd. Nota 5.1.

Nota 5.1: El esquema de la arquitectura está simplificado. Están incluidos todos los módulos con sus correspondientes puertos, pero las conexiones entre puertos son ficticias. Una flecha sólida indica que los dos puertos de sus extremos están conectados directamente en la arquitectura real. Sin embargo una flecha punteada, indica que la conexión es ficticia, el puerto situado en el extremo final de la flecha depende del puerto situado en el extremo inicial, pero no se conectan directamente.

La arquitectura que utilizamos con un módulo *ACS*, Add Compare Select, y un *RegisterExchange* es muy habitual en los decodificadores Viterbi. Pueden consultarse ejemplos con la misma arquitectura en las referencias *2.8.3* y en el *apartado 2.7*.

El módulo ACS se encarga de ir situando los datos de entrada:
$x=Din_1$ & Din_0=BitsInDecoder(1:0) en la malla trellis. En total necesitamos 4 señales diferentes para representar la malla:

1. Distance(63:0)(6:0) → Distancia hasta llegar a cada uno de los 64 estados.

2. BitIn_I_Encoder(63:0) → Bit de entrada al codificador convolucional (m[n]) para pasar del estado h al i, o del j al i, dependiendo de cuál haya sido la rama ganadora.

3. LastState0(63:0) → El estado elegido (j ó h), anterior al i. Si es j entonces LastState0(i)= '1', y si es h, entonces vale '0'. Es suficiente codificarlo con un bit porque el estado j siempre es impar y el h par. Comprobación en *tabla 5.2.*

4. InitialState(63:0) → Indica si el estado i está inicializado. Se utiliza para el caso particular de los primeros datos de la trama de entrada. En el *apartado 5.5.3* lo explicamos detalladamente, por eso no aparece en la siguiente figura:

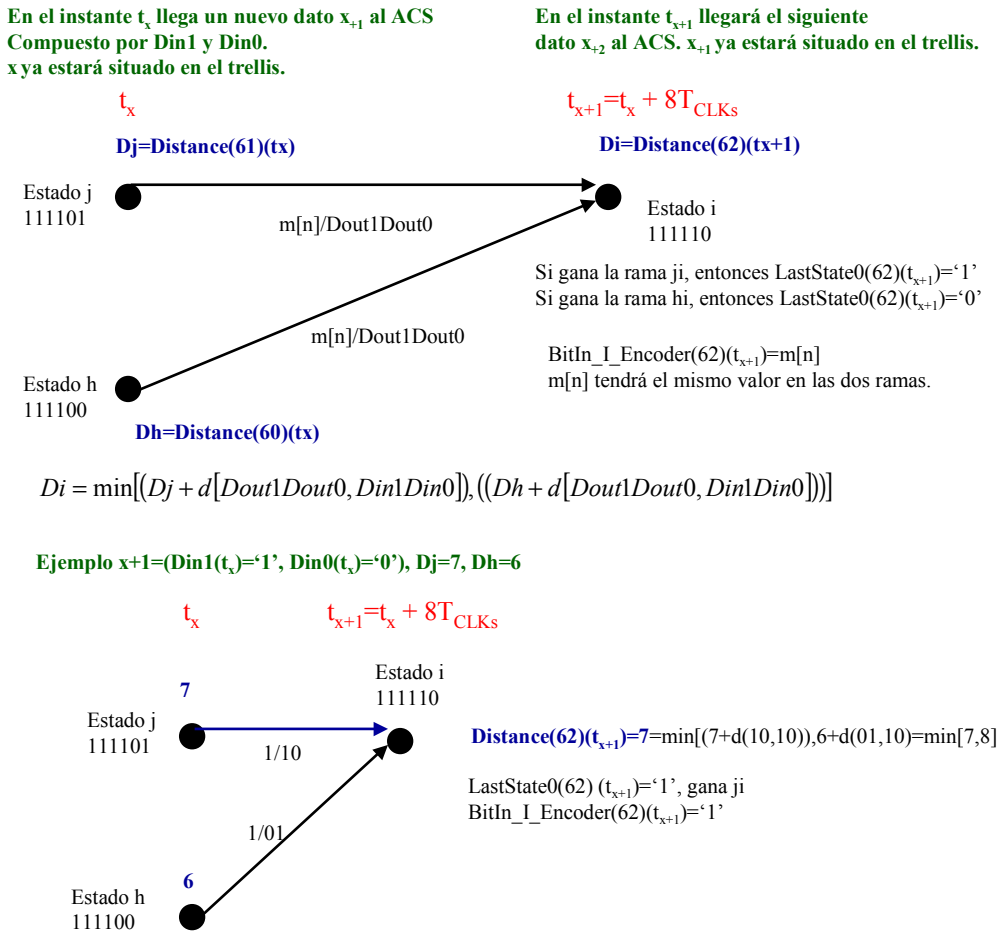


Figura 5.6: Codificación del estado i=62 de la malla trellis.

La notación que empleamos es la misma en todos los apartados:

- A cada dato de entrada compuesto por Din_1 y Din_0 lo llamamos x.
- En el instante t_x llega un nuevo dato x_{+1} al trellis .
- En t_x, el dato x ya está situado en el trellis. Esto significa que Distance(t_x), BitsIn_I_Encoder(t_x), Laststate0(t_x) e InitialState(t_x) contienen la información resultante de situar x en el trellis.
- Un período de proceso después, $t_{x+1}=t_x+8T_{CLKs}$ llega el dato x_{+2}. Y Distance(t_{x+1}), BitsIn_I_Encoder(t_{x+1}), Laststate0(t_{x+1}) e InitialState(t_{x+1}) contienen la información resultante de situar x_{+1} en el trellis.
- El proceso se repite periódicamente, de manera que en $t_{x+2}=t_{x+1}+8T_{CLKs}$ llega un nuevo dato $x_{+3}=(Din_1(t_{x+2}),Din_0(t_{x+2}))$ y x_{+2} ya estará situado en el trellis.
- Tanto en la entrada como en la salida los datos permanecen constantes durante todo el período de proceso, de t_x a $t_x+7\ T_{CLKs}$.

En cada período de proceso, el módulo *ACS* actualiza los 64 estados del trellis. Para ello emplea una combinación de estructuras serie y paralelo. Tiene 8 etapas serie y en cada una de ellas se actualizan 8 estados en paralelo.

RegisterExchange analiza la información procedente del ACS y va almacenando en una memoria el camino superviviente hasta cada uno de los estados. En cada período de proceso actualiza esos caminos supervivientes, desplazándolos una unidad de tiempo en el trellis. Únicamente se almacenan en la memoria los bits m[n], obtenidos de BitIn_I_Encoder. Las otras 3 señales se utilizan para gestionar la memoria.

Un camino superviviente es la secuencia de bits de entrada al codificador m[n], que hay que seguir para desplazarse en el trellis desde la primera posición $t=t_{x-(DecodingDepth-1)}$, dato $x_{-(DecodingDepth-1)}$ situado, hasta la última $t = t_x$ dato x situado. Por tanto cada cada camino está formado por decoding depth bits.

Cuando llega un nuevo dato al trellis, los caminos se desplazan, de manera que el primer bit pasa a ser el de la posición $t=t_{x-(DecodingDepth-2)}$ y el último el de t_{x+1}.

También encuentra de entre los 64 caminos supervivientes cuál es el ganador en cada período de proceso, y obtiene el bit resultado de todo el proceso de decodificación, que es el que debe salir en Data_Out. Ese bit decodificado final es el primer bit del camino superviviente ganador.

5.4.1 Ejemplo con 4 estados.

Para comprender la función de cada uno de los bloques, usamos un sencillo ejemplo con un decodificador de 4 estados, código convolucional 101, 111 (binario) y decoding depth = 5. Trabajamos con el ejemplo del *apartado 2.7*, y nos basamos en la bibliografía citada en *2.8.2*.

El funcionamiento del ViterbiDecoder con 64 estados es equivalente, únicamente hay que utilizar señales con 64 filas en vez de con 4, y utilizar el decoding depth adecuado.

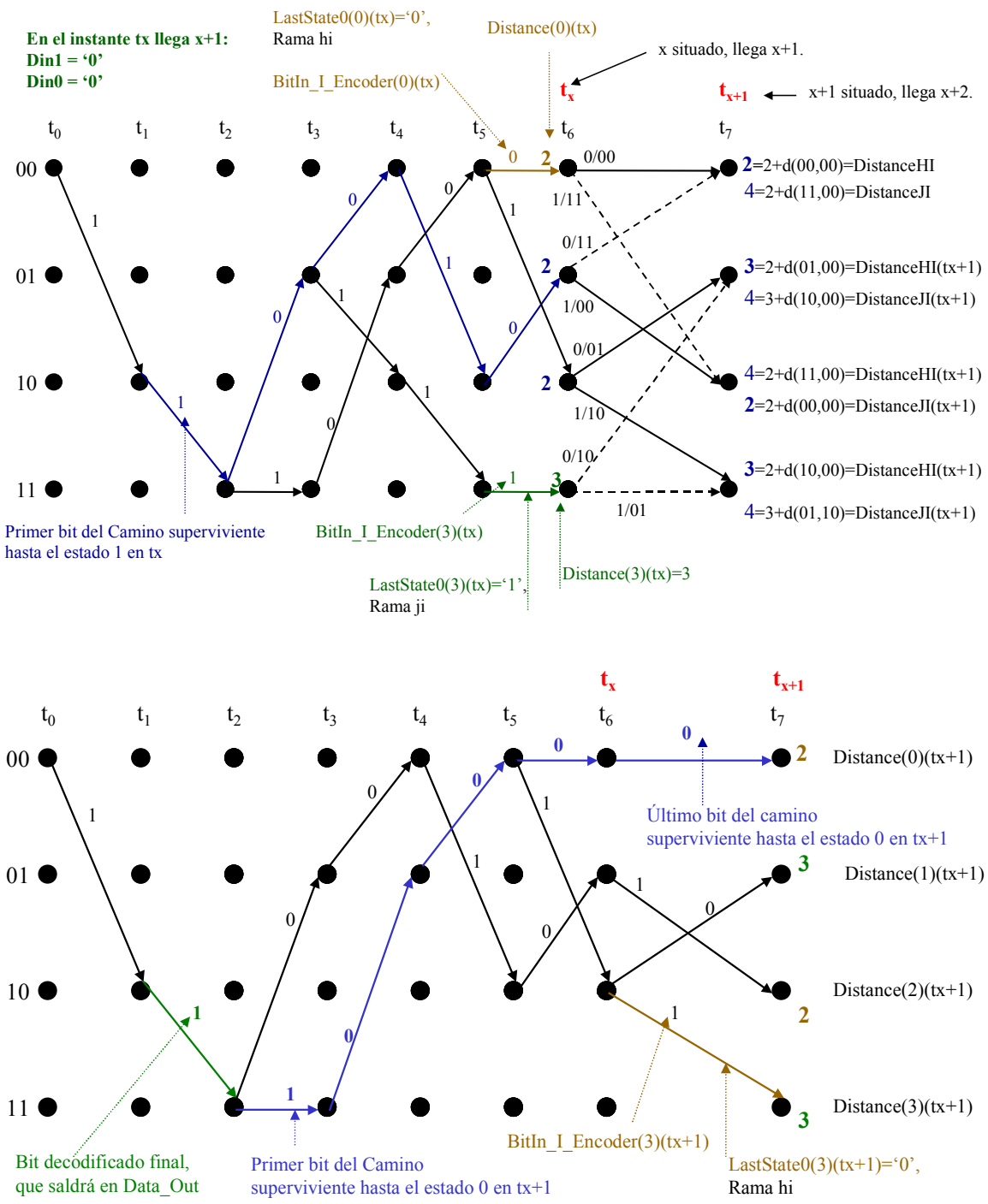


Figura 5.7: Malla trellis del ejemplo con 4 estados

Tabla 5.3: Recopilación de la información del trellis.					
Señal	t_x	$t_{x+1}=t_x+8T_{CLKs}$	Caminos supervivientes		
Distance(3:0)	(3,2,2,2)	(3,2,3,2)	4→último bit	t_x	t_{x+1}
BitIn_I_Encoder(3:0)	('1','1','0','0')	('1','1','0','0')	0→primer bit	43210	43210
LastState0(3:0)	('1','0','0','0')	('0','1','0','0')	Estado(0)	00011	**00001**
InitialState(3:0)	('1','1','1','1')	('1','1','1','1')	Estado(1)	**01001**	01001
BitsInDecoder(1:0)	('0','0')		Estado(2)	10011	10100
			Estado(3)	11101	11001

5.4.2 Descripción de ACS mediante el ejemplo.

Nota 5.2: Para codificar el módulo ACS nos basamos en las citas del *apartado 2.8.4*.

Las señales se actualizan durante el período de proceso que transcurre entre t_x y t_x+7T_{CLKs}. De manera que al llegar el siguiente período de proceso, t_{x+1}, las señales ya estarán actualizadas.

Para actualizar la señal Distance están los módulos *DistanceJIandHI* y *DistanceLastStateBitIn*:

- *DistanceJIandHI* obtiene las distancias hasta llegar a i en el instante t_{x+1} desde los estados j y h en t_x, DistanceJI(t_{x+1}) y DistanceHI(t_{x+1}).

- *DistanceLastStateBitIn* obtiene la distancia hasta llegar a i en t_{x+1}, que será min(DistanceJI(t_{x+1}), DistanceHI(t_{x+1})).

- La distancia en t_{x+1} depende de la distancia en t_x, la distancia en t_{x+2} depende de la distancia en t_{x+1}, y así sucesivamente. Por tanto hay que hacer una realimentación, la señal Distance se realimenta a la entrada InitialDistance del módulo DistanceJIandHI. De esta manera en t_x InitialDistance valdrá (3,2,2,2)=Distance(t_x) y a partir de ella se puede obtener Distance(t_{x+1}).

- Esta realimentación aumenta mucho el área y es lo que da una mayor complejidad al desarrollo y comprensión de la arquitectura.

- Cada módulo calcula la distancia hasta un estado. Entonces necesitamos 64 llamadas a este módulo, una por cada estado Esto nos permite realizar con facilidad diversas variantes de estructuras serie y paralelo.

BitIn_I_Encoder y LastState0 se actualizan en el módulo DistanceLastStateBitIn. Tras realizar el cálculo min(DistanceJI(t_{x+1}),DistanceHI(t_{x+1})), se determina cuál es la rama ganadora: ji ó hi. Dependiendo del resultado, el estado anterior a i en t_{x+1} será el j ó el h. Y BitIn_I_Encoder(i) será el m[n] de la rama ganadora, ji ó hi.

InitialState se actualiza en el módulo *UpdateStateInit*. Se utiliza en el inicio de la trama para situar el primer dato de entrada en el estado cero de la malla trellis. InitialState(t_{x+1}) depende de InitialState(t_x), por eso la señal debe realimentarse y conectarse al puerto InitialState de UpdateStateInit,

Normalize se utiliza para normalizar las distancias cada NormalizeTime períodos de proceso. Esto evita que se produzca un overflow en el caso de que las distancias crezcan indefinidamente. El módulo analiza las 64 distancias del vector Distance, y encuentra la menor de ellas mediante *FindMinIndex* y *Comparator4In*. La normalización consistirá en restar a las 64 distancias del vector Distance, la distancia mínima, señal MinDistance. Para ello *Normalize* envía las señales Normalize_Time y MinDistance al módulo DistancelastStateBitIn, que es el que realiza la resta cuando toca normalizar.

InitialState y Normalize son para casos particulares y por eso no tienen representación en el ejemplo.

5.4.3 Descripción de RegisterExchange mediante el ejemplo.

RegisterExchange controla la memoria del decodificador y obtiene un bit decodificado en cada período de proceso. Para implementarlo nos basamos en la bibliografía citada en *2.8.5*.

Con el ejemplo únicamente tratamos el caso general, en el que la memoria interna está llena y llega un dato nuevo al decodificador en cada período de proceso. Hay otros dos casos, uno en el que al inicio de la trama aún no está llena la memoria y otro en el que ha finalizado la trama y el decodificador debe sacar los bits que aún mantiene almacenados. En estos casos los módulos trabajan de una forma diferente que analizamos en detalle en el *apartado 5.5.9.*

CalculatePaths es el que almacena en la memoria el camino superviviente hasta cada uno de los estados. Para ello, tiene estos datos de partida, obtenidos de la *tabla 5.3*:

Tabla 5.4: Datos iniciales en el módulo CalculatePaths.			
Señal	$t_{x+1}=t_x+8T_{CLKs}$	Caminos supervivientes	
Distance(3:0)	(3,2,3,2)	4→último bit	t_x
BitIn_I_Encoder(3:0)	(‘1’,’1’,’0’,’0’)	0→primer bit	43210
LastState0(3:0)	(‘0’,’1’,’0’,’0’)	Estado(0)	00011
InitialState(3:0)	(‘1’,’1’,’1’,’1’)	Estado(1)	**01001**
BitsInDecoder(1:0)		Estado(2)	10011
		Estado(3)	11101

A partir de los datos anteriores, obtiene los caminos supervivientes en t_{x1}. Realizando las siguientes operaciones para cada estado i.

1. El último bit del camino superviviente hasta i en t_{x+1} es igual a BitIn_I_Encoder(i)(t_{x+1}).
   Si el estado anterior a i en t_{x+1} es el j, LastState0(i)(t_{x+1})=’1’, entonces los bits (3:0) del camino superviviente hasta i en t_{x+1} coinciden con los bits (4:1) del camino hasta j en t_x. Si el estado anterior fuese el h, entonces coinciden con los bits (4:1) del camino hasta h en t_x:

   a. Estado(i)(4:0)(t_{x+1})=BitIn_I_Encoder(i)(t_{x+1})&Estado(j)(4:1)(t_x), si LastState0(i)(t_{x+1})=’1’.
   b. Estado(i)(4:0)(t_{x+1})=BitIn_I_Encoder(i)(t_{x+1})&Estado(h)(4:1)(t_x), si LastState0(i)(t_{x+1})=’0’.

2. Vemos que la memoria tiene una estructura FIFO, en cada período de proceso se introduce un nuevo bit que ocupará la última posición de los caminos (4), y se desplazan una posición los que ya están. De manera que el primer bit de cada uno de los caminos, posición cero, sale de la memoria.
   Ese bit que sale de la memoria se envía al puerto Path

   a. Path(i) (t_{x+1})= Estado(j)(0)(t_x), si LastState0(i)(t_{x+1})=’1’.
   b. Path(i) (t_{x+1})= Estado(h)(0)(t_x), si LastState0(i)(t_{x+1})=’0’.

Ejemplo, en el estado 1→ BitIn_I_Encoder(1)(t_{x+1})='0' y el estado anterior es el h porque LastState0(1)(t_{x+1}) = '0'. Entonces su estado anterior es el número 2 y Estado(2)(4:1)(t_x)=1001. *Nota 5.3*.

- Entonces Estado(1)(t_{x+1})='0'&"1001"="01001".
- Path(1) (t_{x+1})= Estado(2)(0)(t_x)='1'

Nota 5.3: Sabemos que el estado anterior al 1 en t_{x+1} es el 2 mirando la *figura 5.7*. En el caso del ViterbiDecoder de 64 estados es inmediato saber cuál es el estado anterior al i gracias a la constante MatrixLastStates

Partiendo de los datos de la *tabla 5.4* obtendremos el resultado al final del período de proceso:

Tabla 5.5: Datos finales en el módulo CalculatePaths.		
Caminos supervivientes		Señal Path
4→último bit 0→primer bit	t_{x+1} 43210	t_{x+1}
Estado(0)	**00001**=BitIn_I_Encoder(0)(t_{x+1})& Estado(0)(4:1)(t_x)	**Path(0)='1'**=Estado(0)(0)(t_x)
Estado(1)	01001=BitIn_I_Encoder(1) (t_{x+1})& Estado(2)(4:1)(t_x)	Path(1)='1'=Estado(2)(0)(t_x)
Estado(2)	10100=BitIn_I_Encoder(2) (t_{x+1})& Estado(1)(4:1)(t_x)	Path(2)='1'=Estado(1)(0)(t_x)
Estado(3)	11001=BitIn_I_Encoder(3) (t_{x+1})& Estado(2)(4:1)(t_x)	Path(3)='1'=Estado(2)(0)(t_x)

El final de todo el proceso de decodificación lo realiza *ExitDatoOut*. Su trabajo consiste en elegir el bit decodificado final en cada período de proceso.

Ese bit está en el trellis, es el primer bit del camino superviviente ganador. ExitDatoOut recibe de CalculatePaths el primer bit de cada uno de los caminos supervivientes, señal Path. Entonces lo que debe hacer es determinar cuál es el camino ganador y después seleccionar el primer bit de ese camino.

El camino ganador es el que tiene una distancia menor en su última posición. Para determinarlo, ExitDatoOut analiza las 64 distancias que lee del puerto Distance y mediante *FindMinIndex* y *Comparator4IN* obtiene la menor de ellas.

En el ejemplo con el que trabajamos, ExitDatoOut recibiría:

- Distance(3:0)(t_{x+1})=(3,2,3,2) y Path(3:0)(t_{x+1})=('1','1','1','1')

La distancia mínima es 2, correspondiente a los caminos hasta los estados 0 y 2. Por tanto los caminos 0 y 2 son los ganadores, ExitDatoOut elegiría aleatoriamente uno de los dos, y obtendría:
DatosOut(t_{x+1})=Path(2)(t_{x+1})='1' ó **DatosOut(t_{x+1})=Path(0)(t_{x+1})='1'**.

En este caso hemos elegido aleatoriamente el 0 como camino ganador.

5.5 Descripción bloques.

5.5.1 Puertos comunes.

Algunos puertos son comunes en varios de los módulos. Para que la descripción no sea redundante los explicamos solamente en este punto, porque tienen la misma función en todos los bloques.

- Puertos con la información para codificar el trellis. Están descritos en el *apartado 5.4*:

    1. **Distance(63:0)(6:0)** → Distancia hasta llegar a cada uno de los 64 estados. Lo tienen ACS, RegisterExchange y ExitDatoOut. También lo tienen DistanceJIandHI, Normalize, FindMinIndex y Comparator4IN con el nombre **InitialDistance(63:0)(6:0)**.

    2. **BitIn_I_Encoder(63:0)** → Bit de entrada al codificador convolucional (m[n]) para pasar del estado h al i, o del j al i, dependiendo de cuál haya sido la rama ganadora. Lo tienen ACS y DistanceLastStateBitIn. Se llama **BitInEncoder(63:0)** en RegisterExchange y CalculatePaths.

    3. **LastState0(63:0)** → El estado elegido (j ó h), anterior al i.

    4. **InitialState(63:0)** → Indica si el estado i está inicializado. Se utiliza para el caso particular de los primeros datos de la trama de entrada.

- **clk**: La señal de reloj es la misma para todos los bloques síncronos del sistema. Solamente hay dos bloques asíncronos: DistanceJIandHI y Comparator4IN.

- **ce**: Clock Enable. Se utiliza para compatibilizar el diseño con System Generator. Si ce = '1', no afecta a la actividad del módulo. Si ce = '0', entonces el módulo se para, pero almacena su estado. De manera que cuando ce vuelva a ser uno, la actividad se reanuda correctamente desde el estado almacenado. Es equivalente a una pausa y se puede activar y desactivar en cualquier momento, incluso en medio de una trama.

- $\overline{\text{aclr}}$: Reset asíncrono, activo a nivel bajo. Pone todos los puertos de salida, las señales internas y la memoria interna a cero. Si se activa mientras el decodificador está decodificando una trama, ya no se podrá reanudar la decodificación de la trama. Sólo hay dos módulos sin $\overline{\text{aclr}}$ el Comparator4IN y el *MemoriaRAM* (*ver 5.5.11*).

- **EnableIn**: Indica si hay datos disponibles en la entrada, activo a nivel alto. Está presente en todos los bloques salvo en los DistanceJIandHI y Comparator4IN porque son asíncronos y en el *MemoriaRAM* (*ver 5.5.11*). Esta señal se comporta de dos formas diferentes:

  1. En UpdateStateInit, DistancelastStateBitIn, Normalize, FindMinIndex y CalculatePaths. En este caso EnableIn puede activarse y desactivarse en cualquier momento, sin que ello afecte a la actividad del módulo. Esto se debe a que cuando EnableIn está desactivado, el módulo interpreta que no hay datos en la entrada y para su actividad. Pero mantiene su estado actual por lo que cuando EnableIn vuelva a activarse, reanuda su actividad correctamente desde donde se quedó parado. Por tanto se comporta igual que ce, pero es obligatorio utilizar 2 puertos diferentes en la entrada, porque necesitamos que las señales puedan tener valores diferentes en el mismo instante.

  2. En ACS, RegisterExchange y ExitDatoOut. En este caso EnableIn no puede activarse y desactivarse en cualquier momento, porque el trabajo del módulo no sólo depende de si está activo o no. También influye el tiempo que transcurre entre las transiciones de EnableIn. Por eso la activación y desactivación debe hacerse en momentos concretos, que hemos definido previamente.

- **EnableOut**: Indica que las señales de salida son válidas. Activo a nivel alto. Este puerto lo tienen ACS, UpdateStateinit, RegisterExchange y ExitDatoOut.

En todos los bloques salvo en el MemoriaRAM, los registros tienen $\overline{\text{aclr}}$ y ce. Entonces su representación exacta sería la de la *figura 5.8A*. Pero no utilizaremos la representación exacta en las figuras. Porque incluir todos los detalles del registro sólo aporta información redundante y además complica mucho la comprensión de la arquitectura.

En el caso concreto de ACS, FindminIndex y CalculatePaths tampoco incluimos la señal Enablein en los registros. Porque son módulos muy complejos y eliminamos la información que no es imprescindible. El esquema exacto sería:

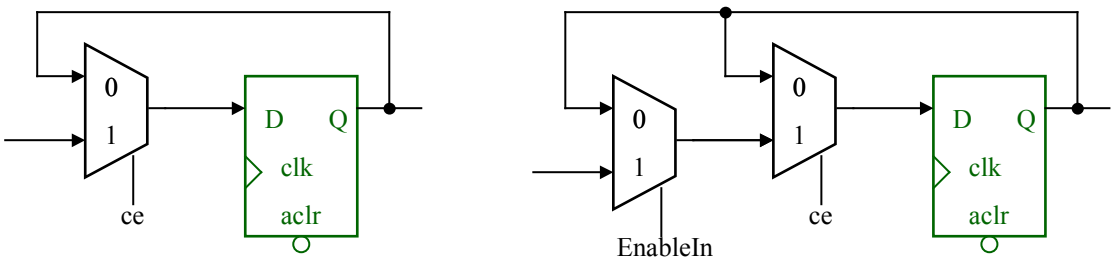


Figura 5.8A: Registros con todas sus señales. Igual para todos los módulos.
Figura 5.8B: Registros con EnableIn en ACS, FindMinIndex y CalculatePaths.

5.5.2 ACS.vhd.

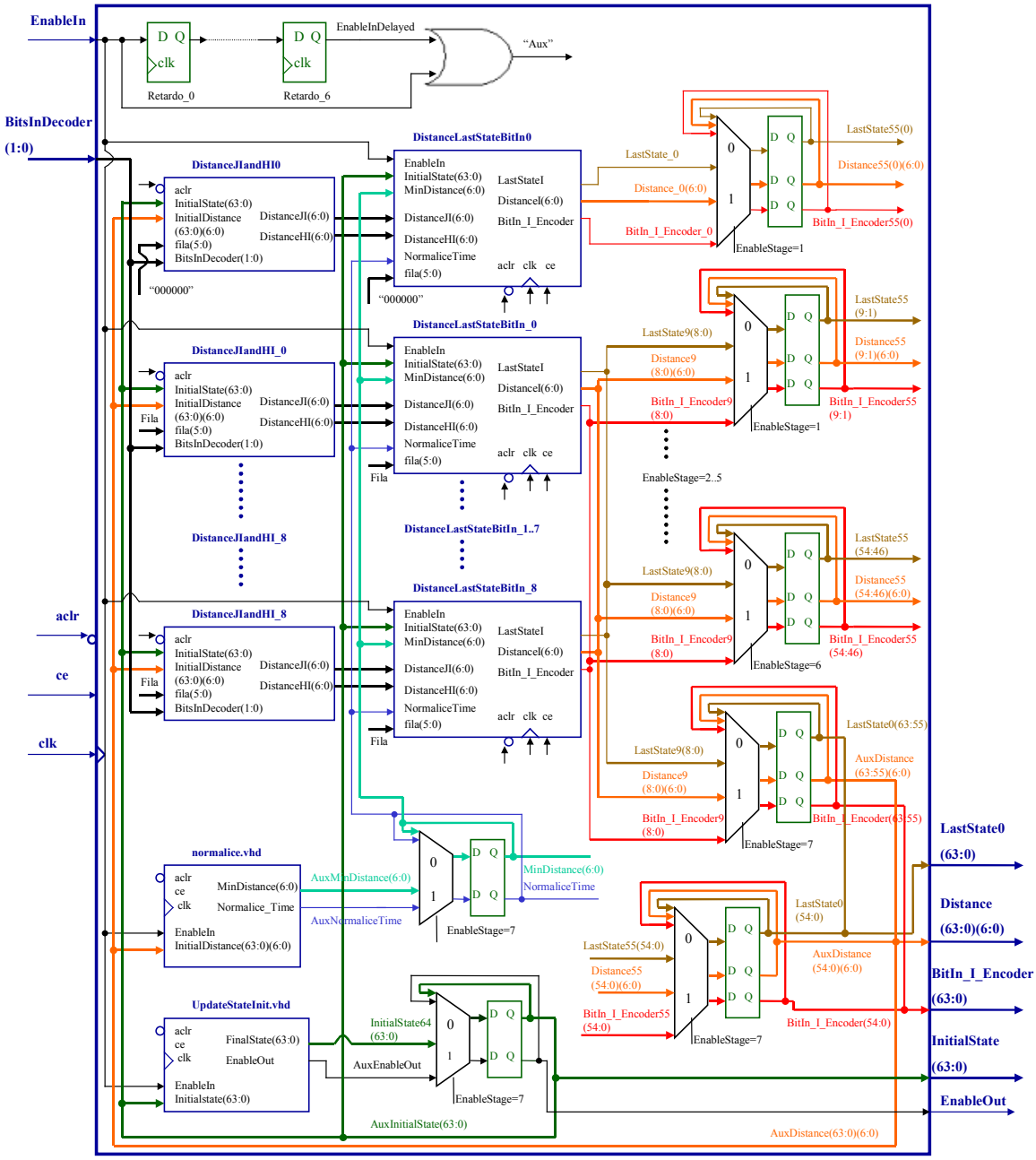


Figura 5.9: ACS.vhd.

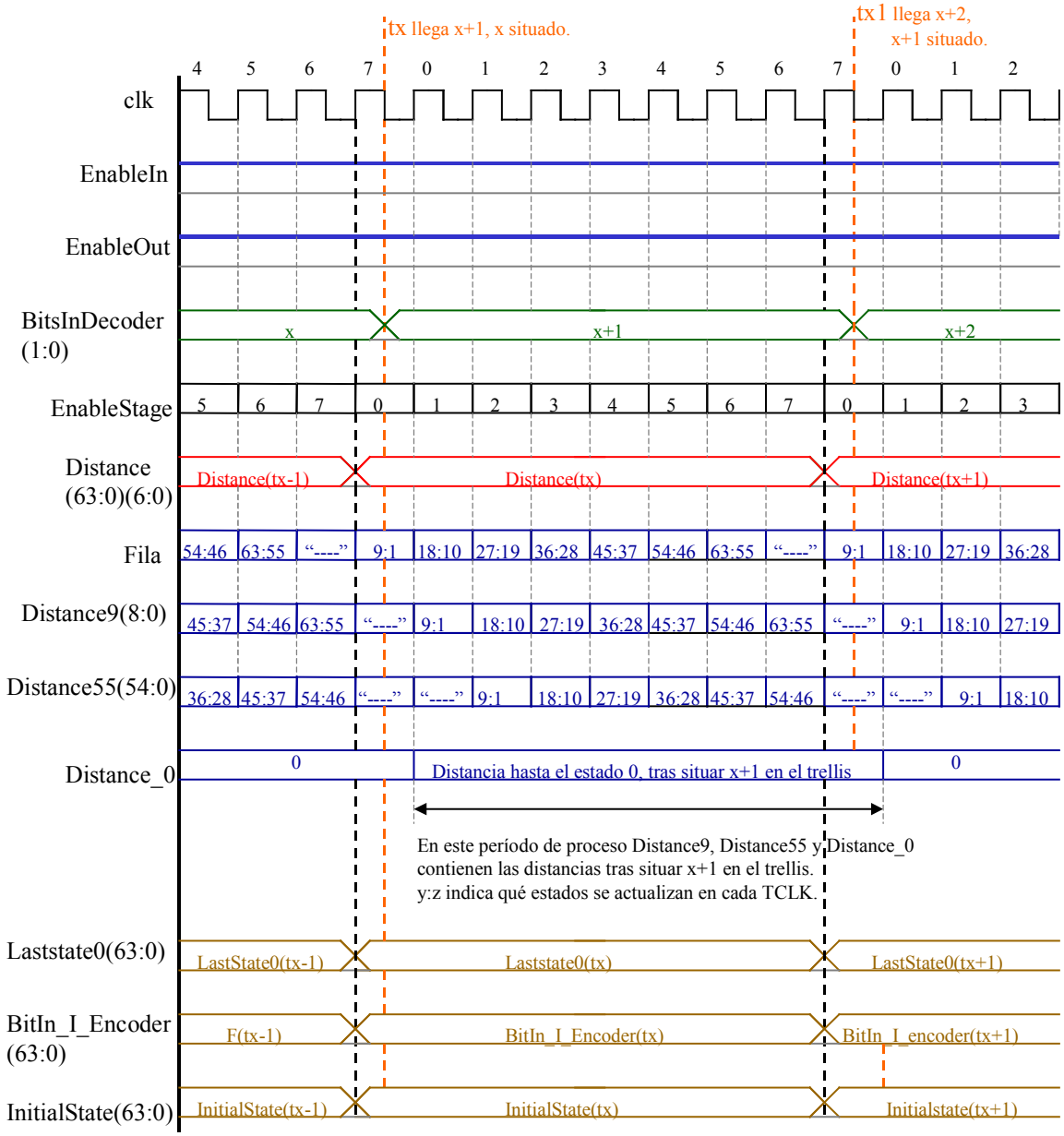
Este módulo es el Add-Compare-Select. Le llegan como entrada los bits de entrada al decodificador:

- BitsInDecoder(1) se corresponde a Dout1 del codificador convolucional (171).
- BitsInDecoder(0) a Dout0 (133).

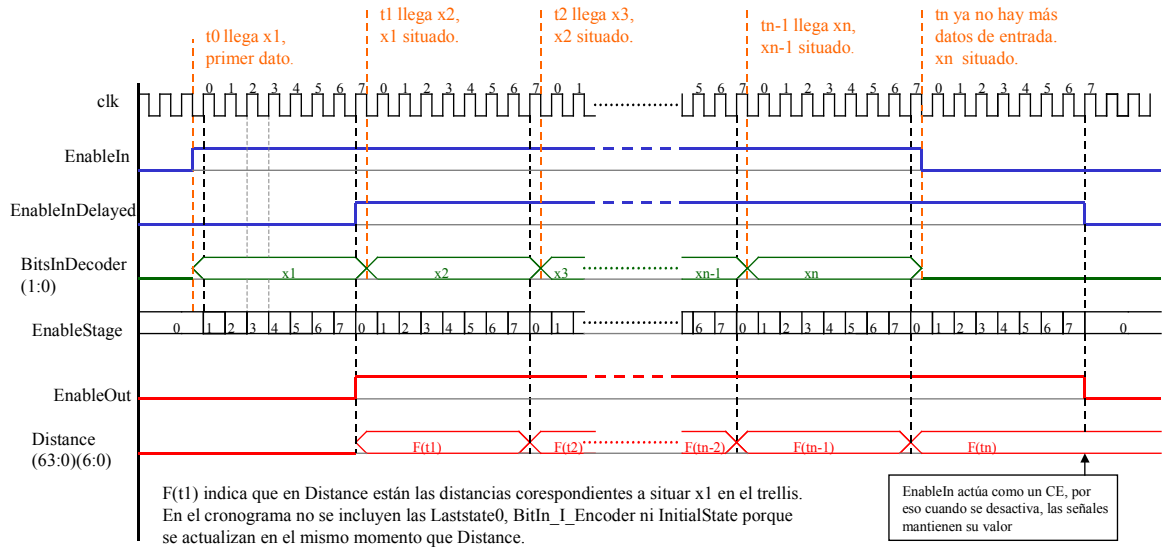
El módulo sitúa cada dato de entrada x=BitsInDecoder(1:0) en el trellis y obtiene las 4 señales que codifican la malla trellis: Distance, BitIn_I_Encoder, Laststate0 e InitialState.
Para implementarlo nos basamos en las referencias del *apartado 2.8.4*.

Esas cuatro señales serán las entradas del módulo RegisterExchange. RegisterExchange las utiliza para gestionar la memoria que contiene decoding depth posiciones de malla trellis. También encuentra los caminos supervivientes, selecciona el ganador y el bit decodificado final, todo ello partiendo de la información que le envía ACS.

La latencia y el período de proceso de un dato en el ACS son $8T_{CLKs}$.



Cronograma 5.4: ACS.vhd Funcionamiento habitual.



Cronograma 5.5: ACS.vhd detalle inicio y final de la trama de entrada.

EnableInDelayed es necesario porque tras t_n aunque ya no haya datos en la entrada, es necesario que EnableStage continúe activo durante un período de proceso más.

En el instante t_x, llega el dato x_{+1} a la entrada del ACS. Y en la salida del bloque tendremos los resultados correspondientes a la colocación del dato x en la malla trellis. Tanto en la entrada como en la salida, los datos permanecen estables de t_x a t_x+7T_{CLKs} (un período de proceso).

En $t_{x+1}= t_x+8T_{CLKs}$, a la entrada llega el dato x_{+2} y en la salida tendremos el x_{+1} situado en el trellis.

Para realizar los cálculos correspondientes al dato x_{+1}, es necesario conocer las distancias y el estado inicial del x en el trellis. Por eso las señales InitialState y Distance, se realimentan de la salida a la entrada.

- Distance se conecta con el puerto InitialDistance de los módulos DistanceJIandHI y Normalize.

- InitialState se conecta con el puerto InitialState de UpdateStateInit, DistanceJIandHI y DistanceLastStateBitIn.

La estructura y la función de los módulos internos está detallada en el *apartado 5.4.2*. No lo repetimos para evitar redundancias.

Una característica muy importante es que el bloque tiene una estructura serie y paralelo. Anteriormente, en los *apartados 5.1 y 5.4* hemos indicado que consta de 8 etapas serie y en cada una de ellas se calculan las señales correspondientes a 8 estados en paralelo. Esto no es exacto, lo expusimos así para no tener que entrar en detalles porque son apartados genéricos. Sin embargo a continuación explicamos detalladamente con cuantos estados se trabaja en cada etapa serie.

La división en estructura serie y paralelo es para calcular las señales Distance, LastState0 y BitIn_I_Encoder. Las tres señales se obtienen con los módulos DistanceJIandHI y DistanceLastStateBitIn.

El proceso para las tres señales es el mismo, así que para simplificar sólo explicamos el caso de la señal Distance.

InitialState se actualiza de manera independiente en el módulo UpdateStateInit. Esta señal no requiere de la división en estructura serie y paralelo.

Actualizar los 64 estados no se puede hacer de la forma más sencilla, (8 estados en paralelo*8 etapas serie) = 64. La estructura real consta de 8 etapas serie, en la primera se actualizan los diez primeros estados. En las seis etapas posteriores se actualizan 9 estados en cada una. De manera que al final de la séptima etapa tendremos 10+6*9=64 estados actualizados. En la última etapa, *ver figura 5.9*, EnableStage=7, lo que hace el módulo es pasar al puerto de salida los 64 estados calculados durante el período de proceso.

La comprensión de la estructura se complica, pero a cambio conseguimos optimizar el compromiso entre área y velocidad.

La estructura consiste en:

- Hay cinco señales Distance_0, Distance9(8:0), Distance55(54:0), AuxDistance(63:0) y Distance(63:0).

- La pareja DistanceJIandH y DistanceLastStateBitIn, actualiza la distancia hasta uno de los 64 estados del trellis. Con una latencia de un TCLK.

- El bloque ACS consta de 8 etapas serie, EnableStage=0..7. Cada etapa consta de un TCLK.

- En la etapa 0 trabajamos con los 10 primeros estados del trellis. Entonces necesitamos 10 bloques DistanceJiandHI y 10 bloques DistanceLaststatebitIn en paralelo.

- En las etapas 1...6 trabajamos con 9 estados del trellis. Entonces necesitamos 9 bloques DistanceJiandHI y 9 bloques DistanceLaststateBitIn en paralelo. Por tanto el primer bloque de la estructura paralelo sólo se usa en la etapa cero.

- La señal Fila indica con que estados van a trabajar en cada etapa los módulos DistanceJIandHI y DistanceLastStateBitIn. Estos estados estarán actualizados un TCLK después en las señales Distance_0 y Distance9.

- Los valores de Distance_0 y Distance9 pasan a distance55 en la siguiente etapa.

- En la última etapa, ya están los 64 estados calculados. Los estados (63:55) estarán en Distance9 y los (54:0) en Distance55. En esta última etapa, asignamos los valores finales al puerto de salida Distance y a AuxDistance. De esta manera en el siguiente T_{CLK}, cuando comienza el siguiente período de proceso, tanto en Distance como en AuxDistance estarán los valores actualizados.

  - Distance(54..0) <= Distance55(54..0);
  - AuxDistance(54..0) <= Distance55(54..0).
  - Distance(63..55) <= distance9(8:0).
  - AuxDistance(63..55) <= distance9(8:0).

Tabla 5.6: Proceso completo en ACS, etapa a etapa

Tiempo	Etapa	Señal Fila.	Estados actualizados en Distance9	Estados actualizados en Distance55	Distancia en el puerto de Salida.
t_x llega x_{+1} x ya está situado en el trellis.	0	9:0	Sin datos Válidos	Sin datos válidos	Distance(t_x)
$t_x + 1T_{CLK}$	1	18:10	9:0[1]	Sin datos válidos	Distance(t_x)
$t_x + 2T_{CLK}$	2	27:19	18:10	Distance55(9:0)(t_{x+1})	Distance(t_x)
$t_x + 3T_{CLK}$	3	36:28	27:19	Distance55(18:10)(t_{x+1})	Distance(t_x)
$t_x + 4T_{CLK}$	4	45:37	36:28	Distance55(27:19)(t_{x+1})	Distance(t_x)
$t_x + 5T_{CLK}$	5	54:46	45:37	Distance55(36:28)(t_{x+1})	Distance(t_x)
$t_x + 6T_{CLK}$	6	63:55	54:46	Distance55(45:37)(t_{x+1})	Distance(t_x)
$t_x + 7T_{CLK}$	7	Nada	63:55	Distance55(54:46)(t_{x+1})	Distance(t_x)
$t_{x+1}..t_{x+1}+7T_{CLK}$ llega x_{+2}. x_{+1} ya está situado en el trellis.	0..7	.....	.....	..... Se van calculando las Distancias correspondientes a x_{+2}	Distance(t_{x+1})
$t_{x+2}..t_{x+2}+7T_{CLK}$ llega x_{+3}. x_{+2} ya está situado en el trellis.	0..7	.....	.....	..... Se van calculando las Distancias correspondientes a x_{+3}	Distance(t_{x+2})

[1] En la etapa cero hay 10 estados. Esto implica que necesitamos actualizar la señal Distance9 y además Distance_0. La señal Distance_0 ya no se vuelve a usar en el resto de etapas.

5.5.3 UpdateStateInit.vhd.

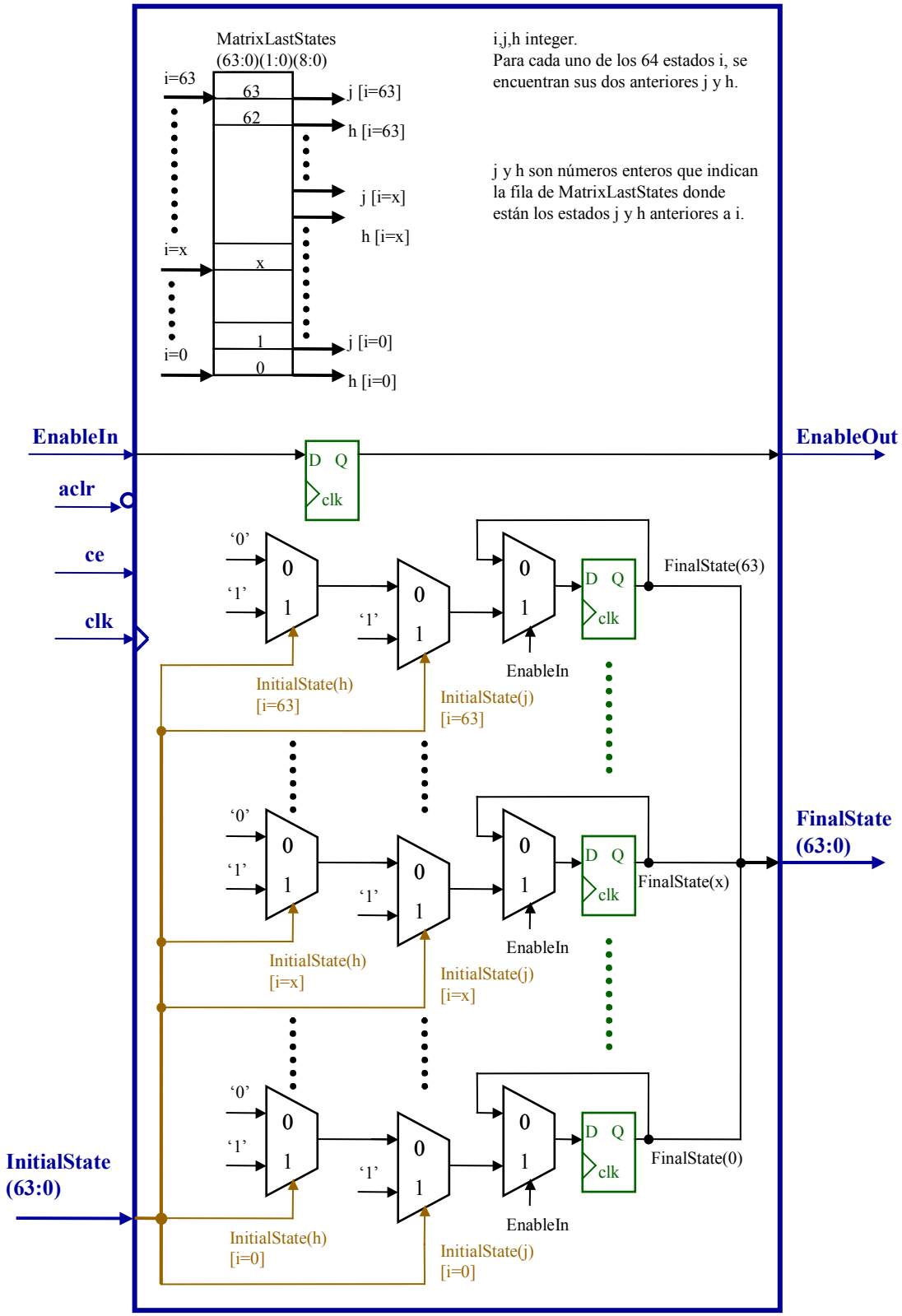


Figura 5.10: UpdateStateInit.vhd.

Actualiza el vector InitialState(63:0) del ACS que indica si un estado en la malla trellis está inicializado o no.

La entrada es un vector con la inicialización en el instante t_x. InitialState(t_x). Y la salida contiene la inicialización en el instante $t_{x+1}=t_x+8T_{CLKs}$. FinalState=InitialState(t_{x+1}).

InitialState(t_{x+1}) depende de InitialState(t_x), por tanto es necesario realimentar la salida a la entrada.

Hay un T_{CLK} de latencia entre EnableIn y EnableOut y entre InitialState y FinalState. Entonces partiendo de InitialState(t_x), un T_{CLK} después en la salida tendremos FinalState=InitialState(t_{x+1}). Sin embargo la señal de salida del módulo ACS, InitialState, debe actualizarse una vez en cada período de proceso, 8 T_{CLKs}. Además entre el EnableIn y el EnableOut del ACS debe haber una latencia de 8 T_{CLKs}. Por eso, cuando el módulo UpdateStateInit haya actualizado FinalState y EnableOut, se mantendrán durante un tiempo antes de asignarlos a la salida del ACS. *Ver figura 5.11*.

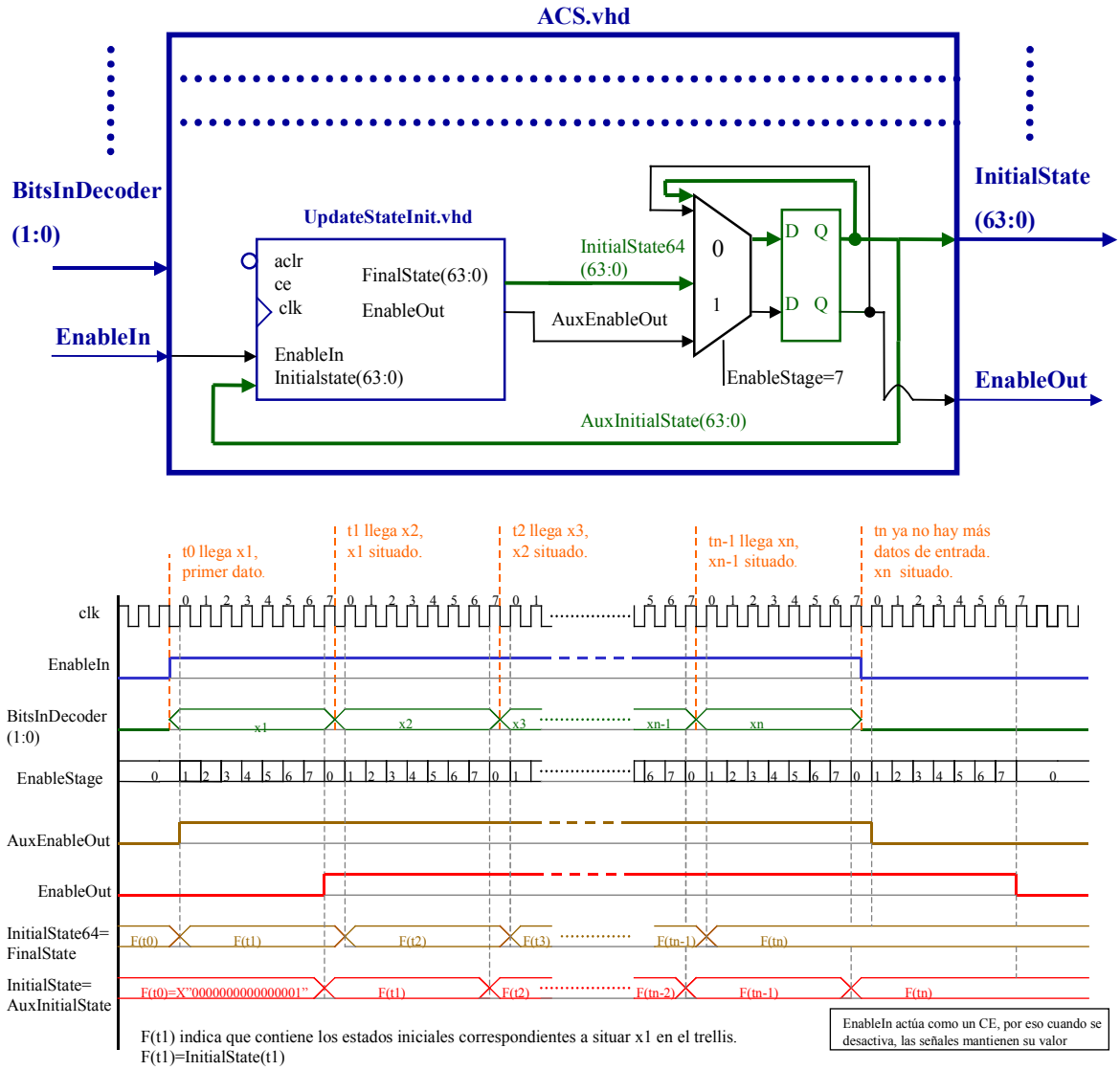


Figura 5.11: Función de UpdateStateInit en ACS.

Utilidad de la señal InitialState. Ejemplo con Viterbi de 4 estados.

Se utiliza para el caso particular de los primeros datos de la trama de entrada.

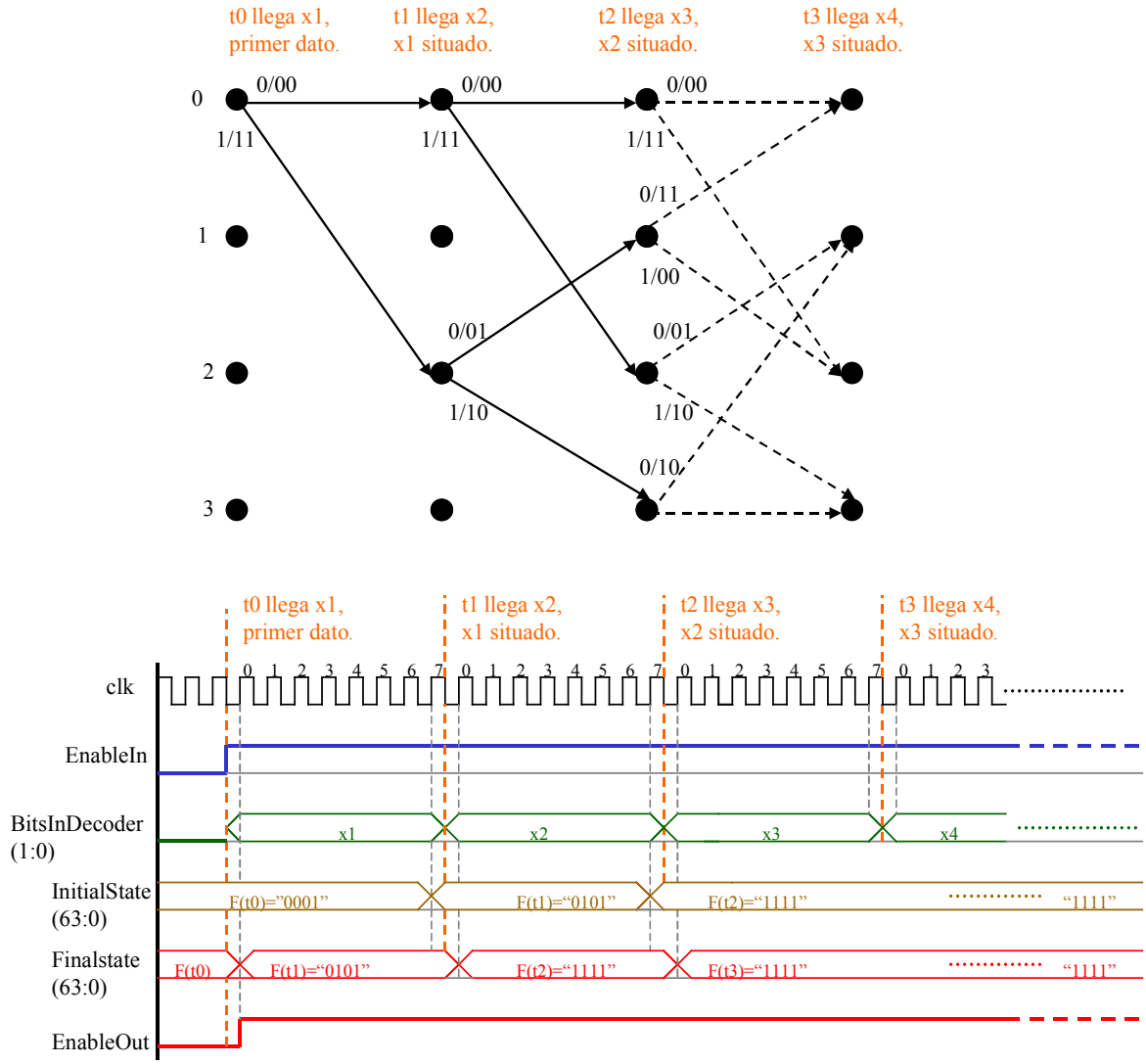


Figura 5.12: Funcionamiento en ejemplo de 4 estados.

La estructura trellis que utiliza el decodificador Viterbi se basa en una malla que tiene 4 estados en el instante t_x. Y se cumple que de todo estado i en t_x parten dos posibles ramas hasta l y m en t_{x+1}.

Pero para que el decodificador Viterbi converja, hay que iniciarlo desde el estado cero. Esto significa que en t_0 cuando llega el primer dato x_1 de la trama de entrada, sólo hay un estado inicializado el cero, y dos posibles ramas para pasar al instante t_1, las que parten de cero.

En t_1, cuando llega x_2 habrá dos estados inicializados, los l y m que tienen como anterior el cero, estos estados son el 0 y el 2 en la de 4 estados. Habrá dos caminos supervivientes los que llegan hasta los estados l y m. Y para pasar al instante t_2 parten cuatro ramas, dos ramas de l y dos de m.

De esta manera en t_2 ya están los 4 estados inicializados, y habrá acabado el proceso. Los estados permanecerán inicializados durante el resto de símbolos de la trama de entrada.

Una opción para que el Viterbi parta del estado cero en t_0 es iniciar la trama de entrada con una cadena de ceros. Porque esa palabra hace converger la malla al estado cero. Ver *apartados 2.5 y 2.6.* Al finalizar la cadena de ceros, el estado ganador será el cero y de él partirán los caminos supervivientes

Sin embargo nosotros no utilizamos esta opción porque limita mucho al decodificador, ya que obliga a que el codificador inicie las tramas con una cadena de ceros. Nuestro decodificador acepta tramas que contengan únicamente símbolos de datos, porque realiza la inicialización de la malla con la señal InitialState:

InitialState(3:0) indica que estados están inicializados y lo actualiza UpdateStateInit en cada período de proceso:

- Un estado i está inicializado cuando la casilla i del vector InitialState vale '1'. Esta señal es accesible al resto de módulos del decodificador y gracias a ella determinan si hasta un estado i concreto llegan ramas del trellis o no.

- Decimos que un estado i está inicializado cuando pueden partir de él caminos hacia el siguiente período de proceso del trellis.

- El trabajo de UpdateStateInit se basa en que todo estado i que tenga al menos uno de sus 2 estados anteriores (j ó h) inicializados en t_x, estará inicializado en t_{x+1}.

Tabla 5.7: Proceso actualización InitialState. Ejemplo malla 4 estados.
Si $t < t_0$
Aún no han llegado datos al decodificador. En esta situación Initialstate=”0001”. Solamente está inicializada la semilla, el estado cero. EnableIn=’0’ porque no hay datos de entrada, así que UpdateStateInit no trabaja.
$t_0 \leq t < t_1$
Se activa EnableIn y UpdateStateInit comienza a trabajar. En este período de proceso el resto de módulos trabajan con InitialState(t_0)=”0001”, así que parten correctamente del único estado inicializado, el cero. UpdateStateInit actualiza InitialState, de manera que en t_1 valdrá “0101”. Para realizar esta actualización realiza las siguientes operaciones. 1. Para cada estado i encuentra sus dos anteriores j y h. 2. Si j ó h están inicializados en t_0, entonces i se inicializa en t_1. 3. El estado i=0 tiene como anteriores el h=0 y el j=2. Como el h está inicializado, → InitialState(0)(t_1)=’1’. 4. El estado i=1 tiene como anteriores el h=2 y el j=3. Como ninguno de los dos está inicializado en t_0 → InitialState(1)(t_1)=’0’. 5. El estado i=2 tiene como anteriores el h=0 y el j=1. Como el h está inicializado, → InitialState(2)(t_1)=’1’. 6. El estado i=3 tiene como anteriores el h=2 y el j=3. Como ninguno de los dos está inicializado → InitialState(3)(t_1)=’0’.

$t_1 \leq t < t_2$
Cuando llegue el instante t_1, los módulos trabajan con InitialState(t_1)=”0101” y UpdateStateInit actualiza la señal para obtener InitialState(t_2)=”1111”. 1. El estado i=0 tiene como anteriores el h=0 y el j=2. Como h y j están inicializados, → InitialState(0)(t_2)=’1’. 2. El estado i=1 tiene como anteriores el h=2 y el j=3. Como el h está inicializado en t1 → InitialState(1)(t_2)=’1’. 3. El estado i=2 tiene como anteriores el h=0 y el j=1. Como el h está inicializado, → InitialState(2)(t_1)=’1’. 4. El estado i=3 tiene como anteriores el h=2 y el j=3. Como el h está inicializado, → InitialState(3)(t_1)=’1’.
$t \geq t_2$
A partir de t2 ya están todos los estados inicializados, así que InitialState = “1111” y ya no se modificará durante el resto de la trama. Al finalizar la trama y antes de comenzar la siguiente es necesario activar el $\overline{aclr}$ para volver a dejar InitialState en su estado inicial “0001”.

Aplicación en nuestro decodificador con 64 estados.

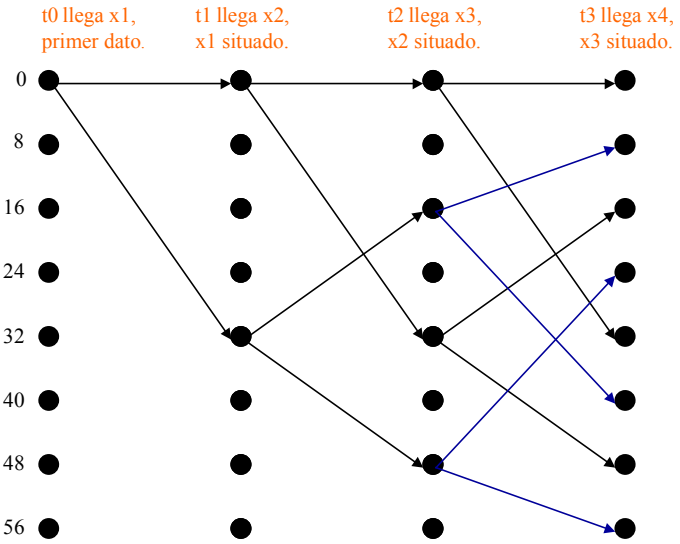


Figura 5.13: Actualización de InitialState en decodificador con 64 estados .

Tabla 5.8: Proceso actualización InitialState. Malla 64 estados.		
	Número de estados inicializados.	InitialState(t)
T≤t0	Sólo el cero, la semilla.	X“0000000000000001”
T=t1	2, el 0 y el 8.	X“0000000100000001”
T=t2	4, el 0, 16, 32, 48.	X“0001000100010001”
T=3	8, el 0, y los múltiplos de 8.	X“0101010101010101”
T=t4	16, el cero y los múltiplos de 4.	X“1111111111111111”
T=5	32, los pares.	X”5555555555555555”
T≥t6	64. Fin de la inicialización.	X”FFFFFFFFFFFFFFFF”

5.5.4 DistanceJIandHI.vhd.

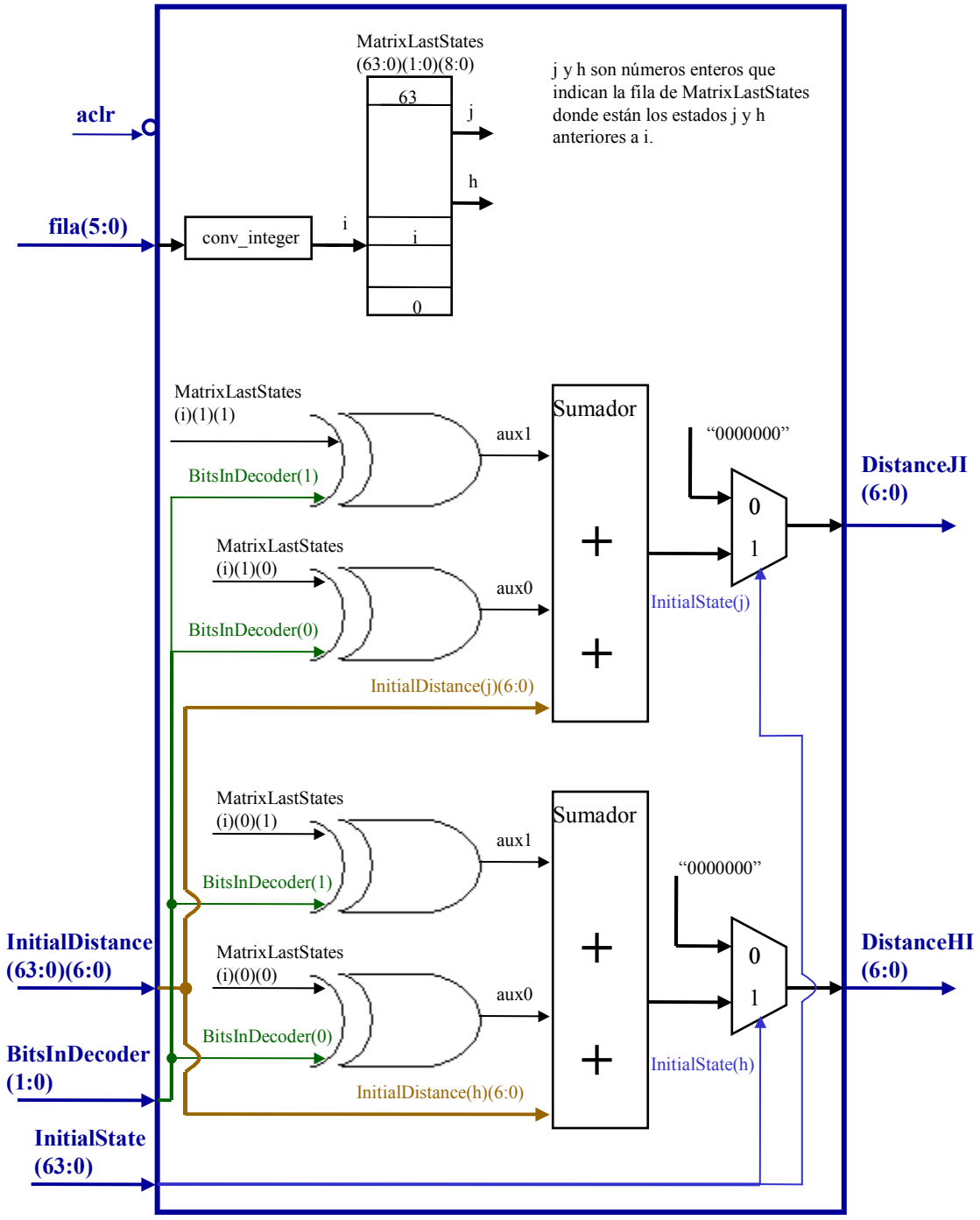


Figura 5.14: DistanceJIandHI.vhd.

Dado un estado i, indicado por la entrada fila, calcula la distancia hasta llegar a él desde sus dos estados anteriores, j y h.

Las entradas del módulo corresponden al instante t_x del trellis y las salidas al t_{x+1}.

- DistanceJI(t_{x+1})=Distance(j)(t_x)+ peso rama ji.
- DistanceHI(t_{x+1})=Distance(h)(t_x)+ peso rama hi.

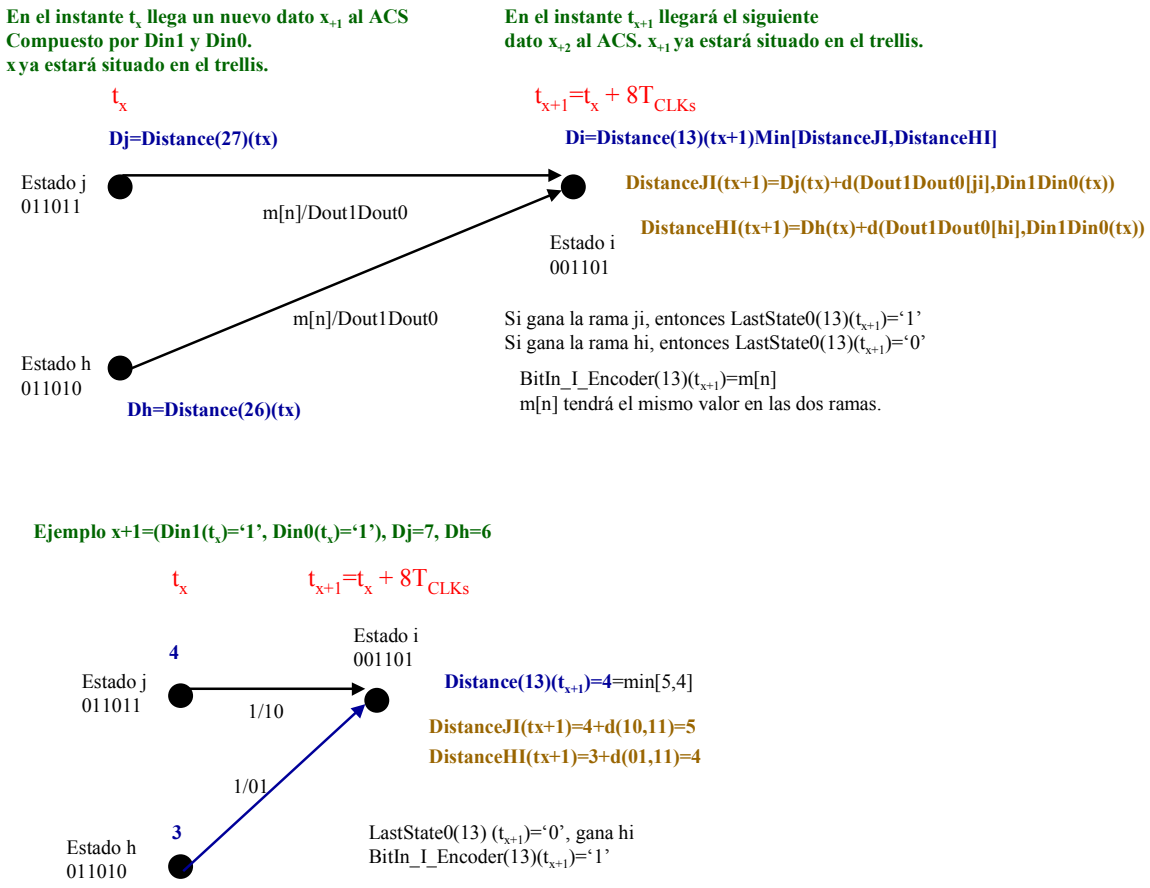


Figura 5.15: Trabajo de DistanceJIandHI y DistanceLastStateBitIn.

El trabajo del módulo consiste en:

1. BitsInDecoder contiene Din1(t_x) y Din0(t_x). InitialDistance contiene las distancias hasta los 64 estados en t_x

2. Encuentra los estados j y h anteriores a i=fila mediante MatrixLastStates.

3. Encontrados j y h, se obtienen de InitialDistance las distancias hasta j y h en t_x.

4. Calcula el peso de las ramas ji y hi. Este peso es la distancia de Hamming entre los bits de entrada al decodificador BitsInDecoder(1:0)=Din_1Din_0 y la cadena $Dout_1Dout_0$ de las ramas ji y hi

5. La distancia final es la suma de los puntos tres y cuatro.

Peso rama ji= d($Dout_1Dout_0$[ji], Din_1Din_0)=(Din_1 xor $Dout_1$[ji]) + (Din_0 xor $Dout_0$[ji]).

Peso rama hi=d($Dout_1Dout_0$[hi], Din_1Din_0)=(Din_1 xor $Dout_1$[hi]) + (Din_0 xor $Dout_0$[hi]).

Cómo sólo calcula las distancias hasta un estado, hay que instanciar este módulo 64 veces. Esto es una ventaja porque da modularidad al diseño, facilitando la división en estructuras serie y paralelo.

Si InitialState(fila)='1' entonces el módulo trabaja normalmente. Si es '0' entonces el estado i no está inicializado, y no hay que calcular las distancias hasta él.

5.5.5 DistanceLastStateBitIn.vhd.

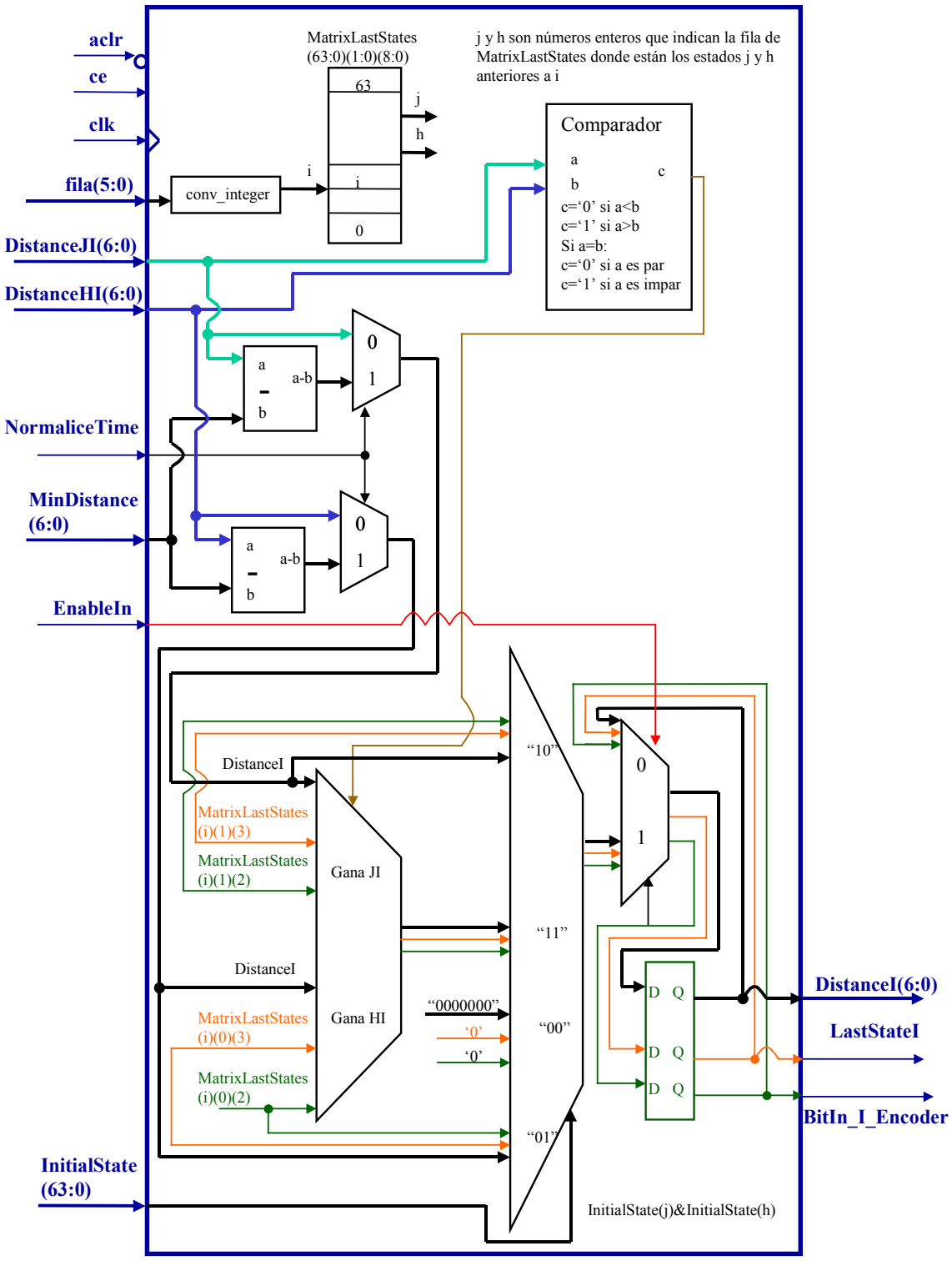


Figura 5.16: DistanceLastStateBitIn.vhd.

Dado un estado i, indicado por la entrada fila, calcula los parámetros necesarios para representar ese estado en el trellis.

El trabajo de este módulo se muestra en la *figura 5.15*. A partir de DistanceJI(t_{x+1}) y DistanceHI(t_{x+1}) obtiene sus salidas:

- DistanceI(t_{x+1}): Es la distancia hasta llegar al estado (i) en t_{x+1}.

- LastStateI(t_{x+1}): Indica cuál es la rama ganadora (ji ó hi) que realiza la transición de t_x a t_{x+1}. LastStateI ='0' indica que gana hi, y '1' que gana ji.

- BitIn_I_Encoder(t_{x+1}): Es el bit de entrada al codificador m[n] que permite la transición de t_x a t_{x+1} en la rama ganadora.

El módulo obtiene los datos anteriores para solamente un estado, así que hay que instanciarlo 64 veces hasta completar la malla. El conjunto de 64 señales de salida se conecta entre sí mediante registros para obtener las señales de salida del ACS: Distance(63:0)(t_{x+1}), LastState0(63:0)(t_{x+1}) y BitIn_I_Encoder(63:0)(t_{x+1}).

El trabajo del módulo consiste en:

1. Obtiene $\text{DistanceI}(t_{x+1}) = \text{Min}[\text{DistanceJI}(t_{x+1}), \text{DistanceHI}(t_{x+1})]$

2. En caso de que la distancia mínima sea la de la rama ji, elige esa rama como ganadora, por tanto LastStateI(t_{x+1})='1'. Si por el contrario el mínimo corresponde a hi, entonces gana esa rama y LastStateI(t_{x+1})='0'.

3. En el caso de DistanceJI=DistanceHI, entonces si DistanceJI es par gana la rama ji, y si es impar gana hi. La especificación del decodificador Viterbi indica que en caso de igualdad, se puede elegir aleatoriamente cualquiera de las 2 ramas, así que el sistema que hemos empleado es correcto.

4. BitIn_I_Encoder toma el m[n] de la rama ganadora.

Podríamos haber implementado un sistema que realizase una decisión más aleatoria en el caso de que DistanceJI=DistanceHI. Pero eso complicaría el diseño sin obtener ningún beneficio.

Otra opción posible sería que en caso de igualdad siempre elijamos una de las 2 ramas. No hemos empleado este sistema, porque no le otorgaría ninguna característica aleatoria a la decisión, mientras que el que hemos elegido tiene cierto carácter aleatorio y la complejidad aunque mayor, es despreciable.

En cualquier caso, la especificación no indica que sistema es mejor para el decodificador, puede utilizarse cualquiera de los 3.

Notas:

- Si Normalize_Time= '1' , entonces ha llegado el momento de normalizar. Lo que se hace es restar al valor que hayamos obtenido en DistanceI el contenido de MinDistance. MinDistance contiene la menor distancia de entre las 64 distancias hasta cada uno de los estados en t_x.

- Si Normalize_Time='0', la señal MinDistance se ignora.

- Si InitialState(fila)='1' el módulo trabaja normalmente. Si es '0' entonces el estado i no está inicializado, y no hay que calcular nada.

- Si EnableIn vale cero las salidas mantienen su valor. Cuando EnableIn vuelva a '1' se reanuda la actividad normal. Por tanto EnableIn actúa como un Clock Enable.

La latencia y el período de proceso de este módulo es 1 T_{CLK}. El DistanceJIandHI tiene la particularidad de que es asíncrono, así que su latencia es cero.

Para obtener las tres salidas de este módulo, hay que utilizar una pareja DistanceJIandHI y DistanceLastStateBitIn en serie. Esta pareja se instancia 64 veces con una estructura de 8 etapas en serie * 8estados en paralelo en cada etapa serie.

La latencia de cada etapa serie es la suma de latencias de DistanceJIandHI y DistanceLastStateBitIn, por tanto es de sólo1 T_{CLK} .

La latencia del ACS es la suma de las latencias de las 8 etapas serie. Tendremos por tanto 8 T_{CLKs}, igual a su período de proceso de un dato.

En caso de que DistanceJIandHi fuese síncrono tendríamos una latencia de 2 T_{CLKs} en cada etapa y habría dos opciones posibles de rediseño para el ACS, las dos negativas:

1. Mantener igual la estructura del ACS. En este caso las prestaciones del decodificador disminuyen, ya que el período de proceso de un dato pasaría a 16 T_{CLKs}. Entonces como F_{CLK} no varía, la frecuencia con la que llegan los datos al decodificador se debe dividir por dos. Además se pierde velocidad manteniendo el mismo área en el diseño.

2. Mantener el período de proceso de 8 T_{CLKs}. De esta manera la frecuencia de llegada de datos al decodificador no varía. Pero el inconveniente es que debemos utilizar una estructura de 16 estados en paralelo *4 en serie. Así que el área aumenta.

5.5.6 Normalize.vhd.

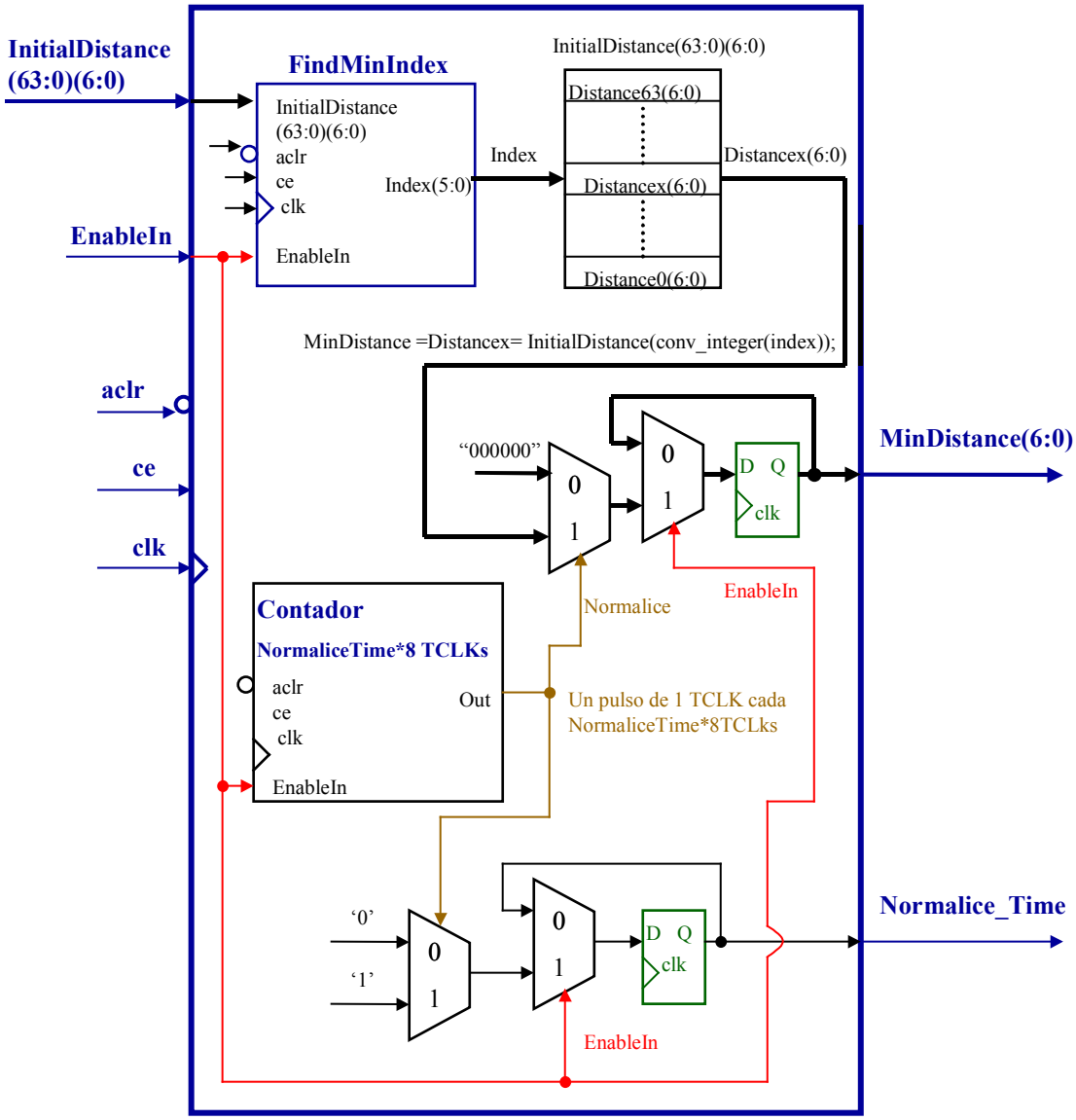


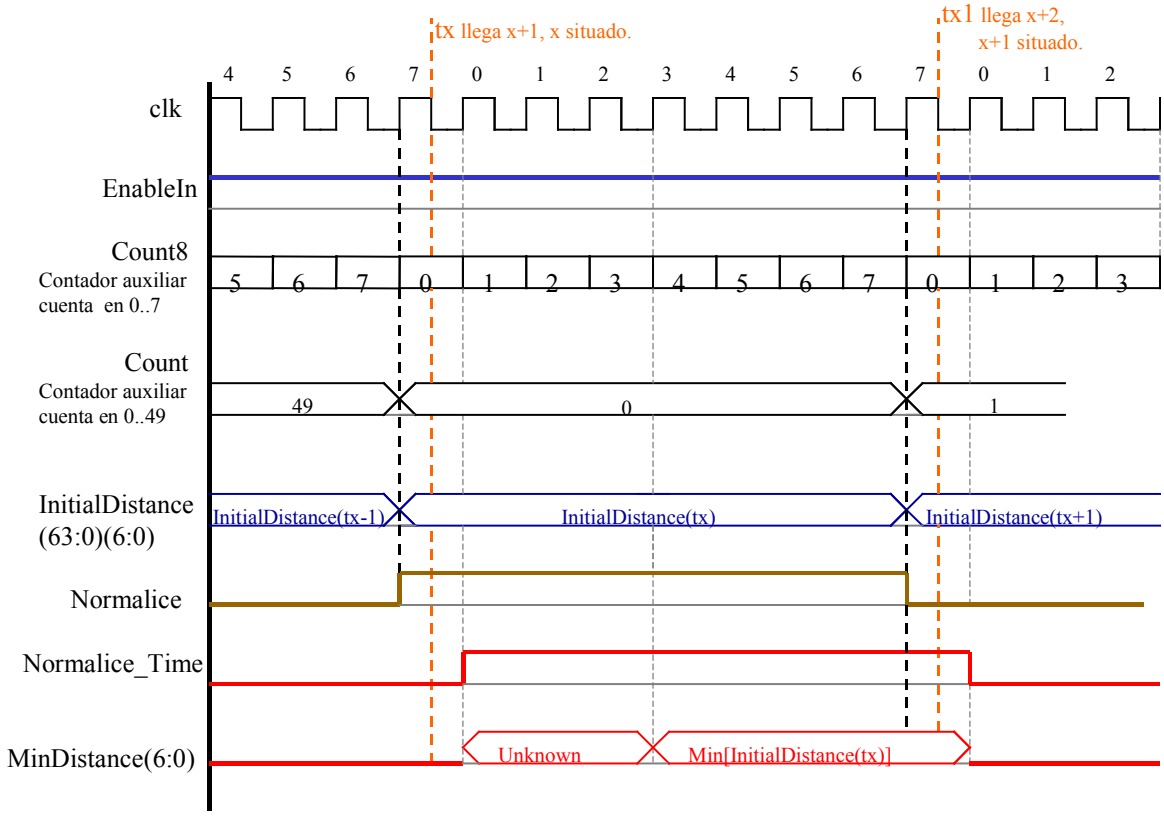
Figura 5.17: Normalize.vhd.

$$\begin{cases} \text{MinDistance} = \min\limits_{0 \le i \le 63} \left[\text{InitialDistance}(i)\right] \Rightarrow \text{Si Normalice\_Time} = '1' \\ \text{MinDistance} = "0000000" \Rightarrow \text{Si Normalice\_Time} = '0' \end{cases}$$

Cada NormalizeTime períodos de proceso se normalizan las 64 distancias presentes en la malla trellis, restándoles la distancia mínima que haya obtenido este módulo. Con esto se evita que crezcan indefinidamente, y hace imposible que se produzca un overflow. Información sobre las distintas técnicas de normalización en: [6], [7] y [8].

Para ello Normalize_Time se activa, a '1', durante un período de proceso, 8_{TCLKs}, cada NormalizeTime períodos de proceso. Y permanece a cero durante NormalizeTime – 1 períodos.

Cuando Normalize_Time esté activo, MinDistance contendrá la distancia mínima de entre las 64 que hay hasta llegar a cada uno de los estados en t_x. Esas 64 distancias iniciales estarán en el puerto InitialDistance. Si Normalize_Time vale '0', entonces MinDistance="0000000". De esta forma evitamos transiciones innecesarias.



Cronograma 5.6: Normalize.vhd. NormalizeTime=50.

Las dos salidas de este módulo se conectan con el DistanceLastStateBitIn, que es el que se encarga de restar a las distancias hasta los 64 estados el valor MinDistance. Las dos señales deben estar estables en la entrada de DistanceLastStateBitIn durante un período de proceso completo. Para conseguirlo utilizamos un registro en la salida del bloque Normalize, ver *figura 5.18*.

Entonces MinDistance tiene un latencia total de un período de proceso. Esto significa que en la entrada de DistanceLastStateBitIn tendremos:

- MinDistance que contiene la distancia mínima en t_x.
- InitialDistance en DistanceJIandHI que corresponde a t_{x+1}.
- La normalización consiste en $\sum_{i=0}^{63}\left[\text{Distance(i)}(t_{x+1}) - \min_{0\le i\le 63}\left[\text{Distance(i)}(t_x)\right]\right]$

A pesar del desfase no hay posibilidad de error porque con el paso del tiempo la distancia mínima puede aumentar o permanecer constante, pero nunca disminuir. Así que en la resta nunca obtendremos un valor menor que cero.

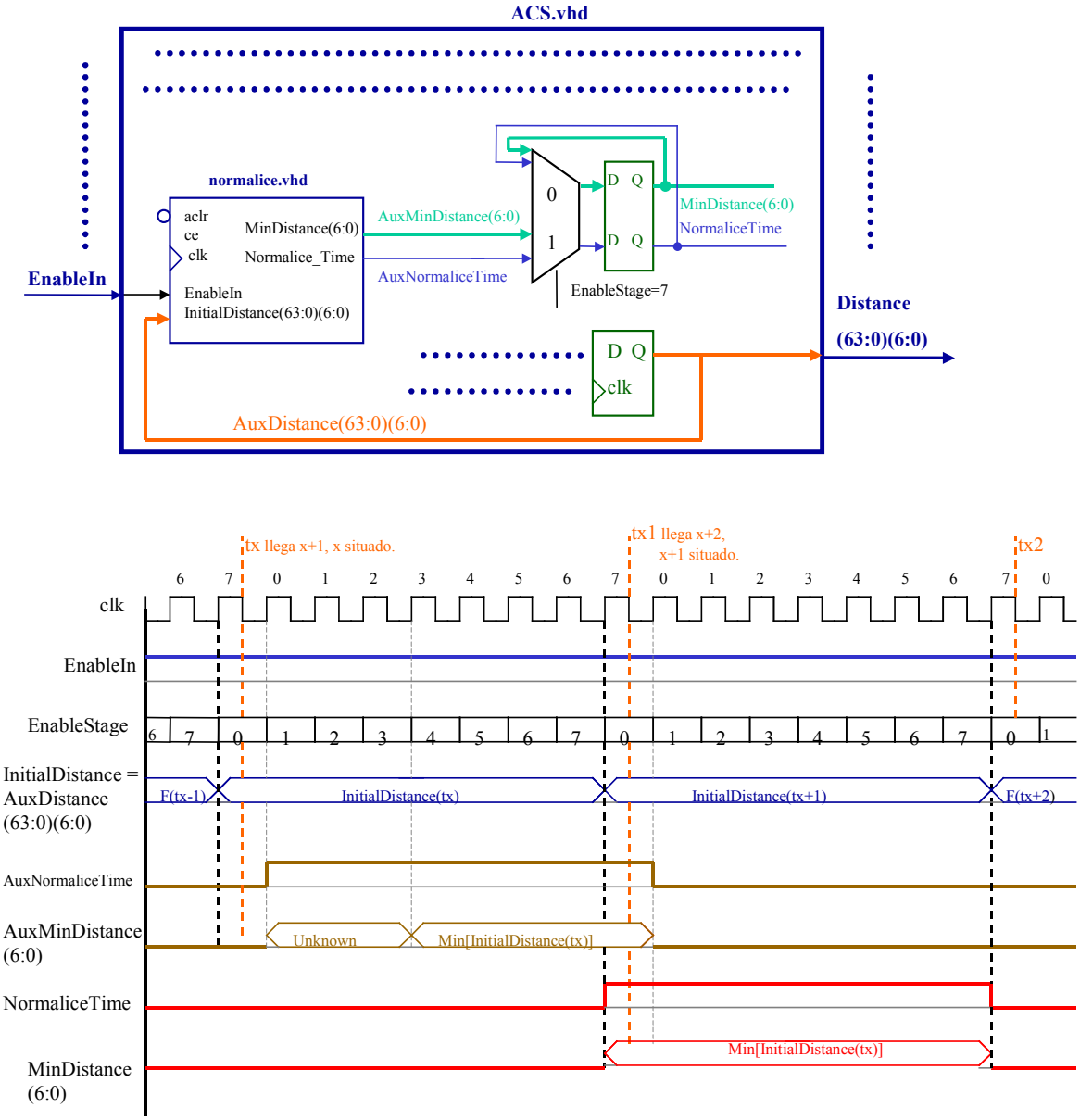


Figura 5.18: Detalle de Normalize integrado en ACS.

Las distancias se codifican con 7 bits, y en cada período de proceso pueden aumentar en 2 unidades. Por tanto para evitar un overflow debe cumplirse:

$$\text{NormaliceTime} \le \frac{2^7 - 1}{2} = \frac{127}{2} = 63 \qquad \text{Nosotros utilizamos 50.}$$

Interesa que NormalizeTime sea lo más alto posible para así disminuir el número de transiciones del decodificador. Así la plataforma en que se implemente tendrá un menor consumo y temperatura. En términos de área y velocidad el valor de NormalizeTime no influye, porque siempre se coge el caso peor.

5.5.7 FindMinIndex.vhd.

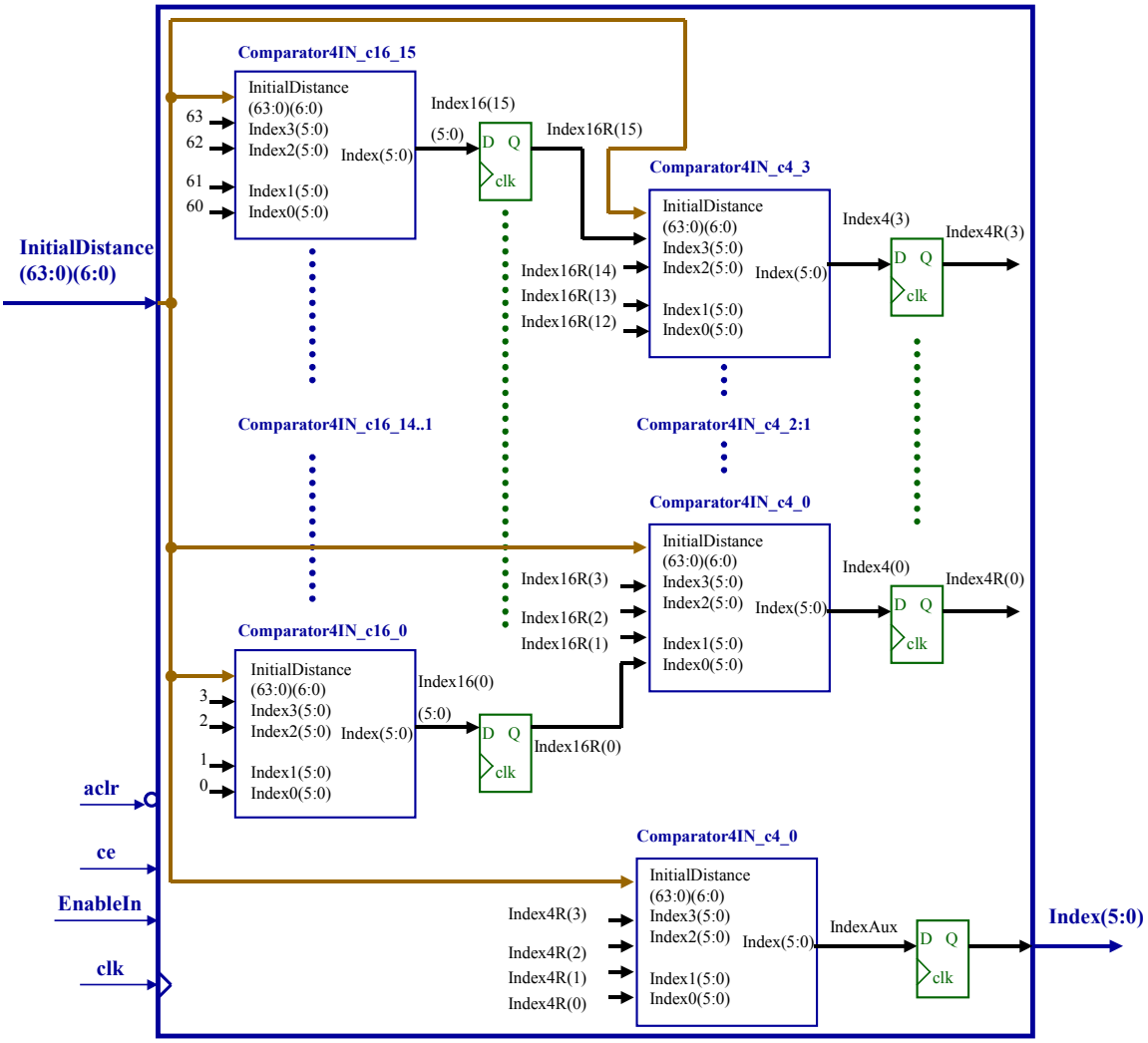


Figura 5.19: FindMinIndex.vhd.

$$\text{Index} = i_x = \min\left[i \in [0..63]\right]_{InitialDistance(ix)=MinDistance=\min\limits_{0\le i\le 63}\left[InitialDistance(i)\right]}$$

Index es el índice de la casilla del vector InitialDistance que contiene la distancia mínima.

Tabla 5.9: Funcionamiento FindMinIndex		
Contenido de InitialDistance	Índice	Resulados
D_{63}	i_{63}	$\text{Index} = i_x$ $D_x = \min\limits_{0\le i\le 63}[D_i]$ $D_x = \text{InitialDistance}(i_x)$
…… ……	……	
D_x	i_x	
…… ……		
D_0	i_0	

El bloque se utiliza en dos situaciones:

1. Para obtener la distancia mínima en el bloque Normalize:
   MinDistance = InitialDistance(conv_integer(Index))=D_x.

2. En ExitDatoOut para elegir el bit ganador al final de todo el proceso de decodificación.

Estructura del bloque:

Se basa en el módulo Comparator4IN. Este módulo tiene en su entrada el vector InitialDistance y 4 índices, Index0..3. Busca las casillas indexadas y obtiene en la salida el índice de la casilla con un valor menor.

Calcula:
Min(InitialDistance(Index0),InitialDistance(Index1),InitialDstance(Index2),InitialDistance(Index3)).
La salida será el índice ganador Index Є Index0..3 tal que InitialDistance(Index) es el mínimo.

El cálculo se hace en tres etapas:

1. La primera etapa consta de 64 índices, por eso se utilizan 16 módulos Comparator4IN en paralelo. El componente cero examina las casillas 0..3 de manera que su salida Index16(0) Є [0..3]. La salida Index16(1) del componente 1 estará en el intervalo [4..7] y así sucesivamente hasta cubrir las 64 casillas, Index16(15) Є [60..63].

2. En la segunda etapa se examinan los 16 índices ganadores de la primera etapa y mediante de 4bloques Comparator4IN obtiene 4 índices ganadores en la salida. De manera que Index4(0) Є [0..15], Index4(1) Є [16..31], Index4(2) Є [32..47] e Index4(3) Є [48..63].

3. En la tercera etapa se obtiene el índice ganador final que es la salida de FindMinIndex. Utiliza un sólo Comparator4IN, que tiene como entradas los índices ganadores en la etapa anterior.

En el caso de que la distancia mínima se repita en varias casillas, este bloque siempre elegirá la que tenga un índice mayor. Cuando buscamos la distancia mínima para el bloque Normalize, da exactamente igual quien gane de entre todos los índices que tengan la misma distancia. Cuando sea el bloque ExitDatoOut el que llama a FindMinIndex, sí que puede haber variaciones entre la elección de un índice u otro. Sin embargo la especificación del Viterbi indica que en esta situación se puede elegir aleatoriamente cualquier camino como ganador. Por tanto nuestro diseño es correcto.

Podría implementarse un sistema en el que la elección tuviese cierto componente aleatorio y no siempre se eligiese esa casilla. Pero eso complicaría el diseño, haría falta más área, consumo y tiempo de operación. Además no es necesaria esa complejidad extra porque la especificación no indica que con este método el decodificador vaya a funcionar mejor.

Notas:

Este bloque es el que limita la frecuencia máxima de trabajo del decodificador, está en el camino crítico. Por eso lo hemos optimizado al máximo. Realizamos el cálculo en tres etapas registradas utilizando la técnica pipeline. Para ampliar información sobre pipeline consultar [9], [10] y [11]. Hemos realizado pruebas con diversas configuraciones, y ésta es con la que mejores resultados hemos obtenido.

Otra opción posible sería que la salida en vez de ser el índice del vector con la distancia mínima, fuese directamente la distancia mínima. Hicimos la prueba y obtuvimos una frecuencia de trabajo menor, por lo que la descartamos.

Tiene una latencia de 3 T_{CLKs}. Los datos deben estar estables en InitialDistance durante esos 3 T_{CLKs}.

Transcurrida la latencia de 3 T_{CLKs}, ya estará en Index el valor correcto. Ese valor permanecerá constante hasta que no se produzca un cambio en InitialDistance.

InitialDistance se mantendrá estable en la entrada durante 8 T_{CLKs}. Por tanto en cada ciclo de proceso, T_{CLKs} 0..7, puede leerse la salida en cualquier instante entre el 3º y el 7º.

En este bloque no hace falta mirar si un estado está inicializado antes de calcular la distancia mínima. Si un estado no está inicializado, en la casilla correspondiente de InitialDistance habrá un cero y se produciría un fallo en este bloque. Sin embargo esto no ocurrirá nunca, porque a ese componente sólo se le llama cuando todos los estados ya estén inicializados.

Se le llama en dos situaciones diferentes:

1. Cuando al final de todo el proceso, última fase del RegisterExchange, hay que buscar el estado con menor peso para elegir el camino hasta él como el ganador. A esta situación se llega después de decoding depth períodos proceso de un dato.

2. Al utilizar el bloque Normalize. Se le llama periódicamente cada NormalizeTime ciclos de proceso.

Al recibir su primer dato, el decodificador solamente tiene el estado 0 inicializado, pero después de 6 períodos de proceso, ya estarán todos inicializados. Por tanto para que el funcionamiento sea correcto, debe cumplirse siempre:

- NormalizeTime >6.
- Decoding depth > 6.

5.5.8 Comparator4IN.vhd.

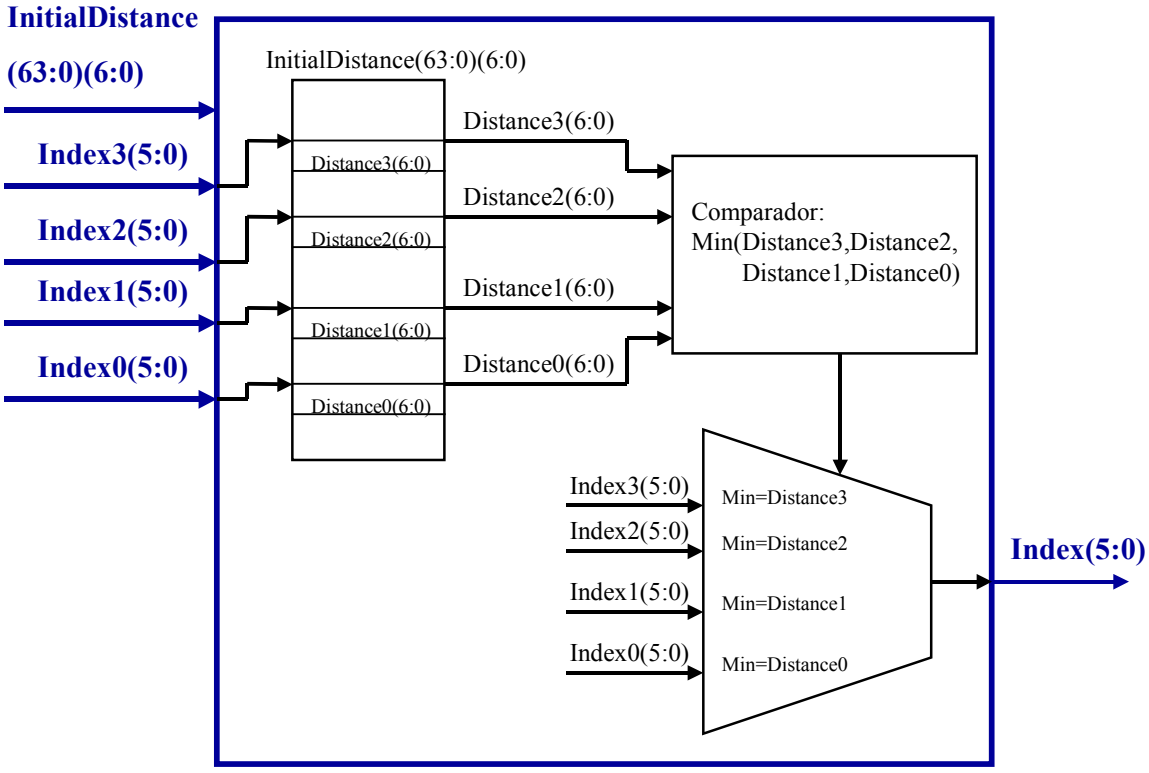


Figura 5.20: Comparator4IN.vhd.

Este módulo tiene en su entrada el vector InitialDistance y 4 índices, Index0..3. Busca las casillas indexadas y obtiene en la salida el índice de la casilla con un valor menor.

- $Distance_0 = InitialDistance(Index_0)$.
- $Distance_1 = InitialDistance(Index_1)$.
- $Distance_2 = InitialDistance(Index_2)$.
- $Distance_3 = InitialDistance(Index_3)$.

- Si $Distance_0=min(Distance_0, Distance_1, Distance_2, Distance_3) \rightarrow Index = Index_0$.
- Si $Distance_1=min(Distance_0, Distance_1, Distance_2, Distance_3) \rightarrow Index = Index_1$.
- Si $Distance_2=min(Distance_0, Distance_1, Distance_2, Distance_3) \rightarrow Index = Index_2$.
- Si $Distance_3=min(Distance_0, Distance_1, Distance_2, Distance_3) \rightarrow Index = Index_3$.

Latencia = 0, no tiene reloj.

En caso de que el valor mínimo sea el mismo en varias de las entradas, se elige la de menor subíndice.

En este bloque no se tiene en cuenta si los estados están inicializados o no. No es necesario hacerlo porque cuando se le llame, sus distancias de entrada siempre estarán inicializadas.

5.5.9 RegisterExchange.vhd. Estructura.

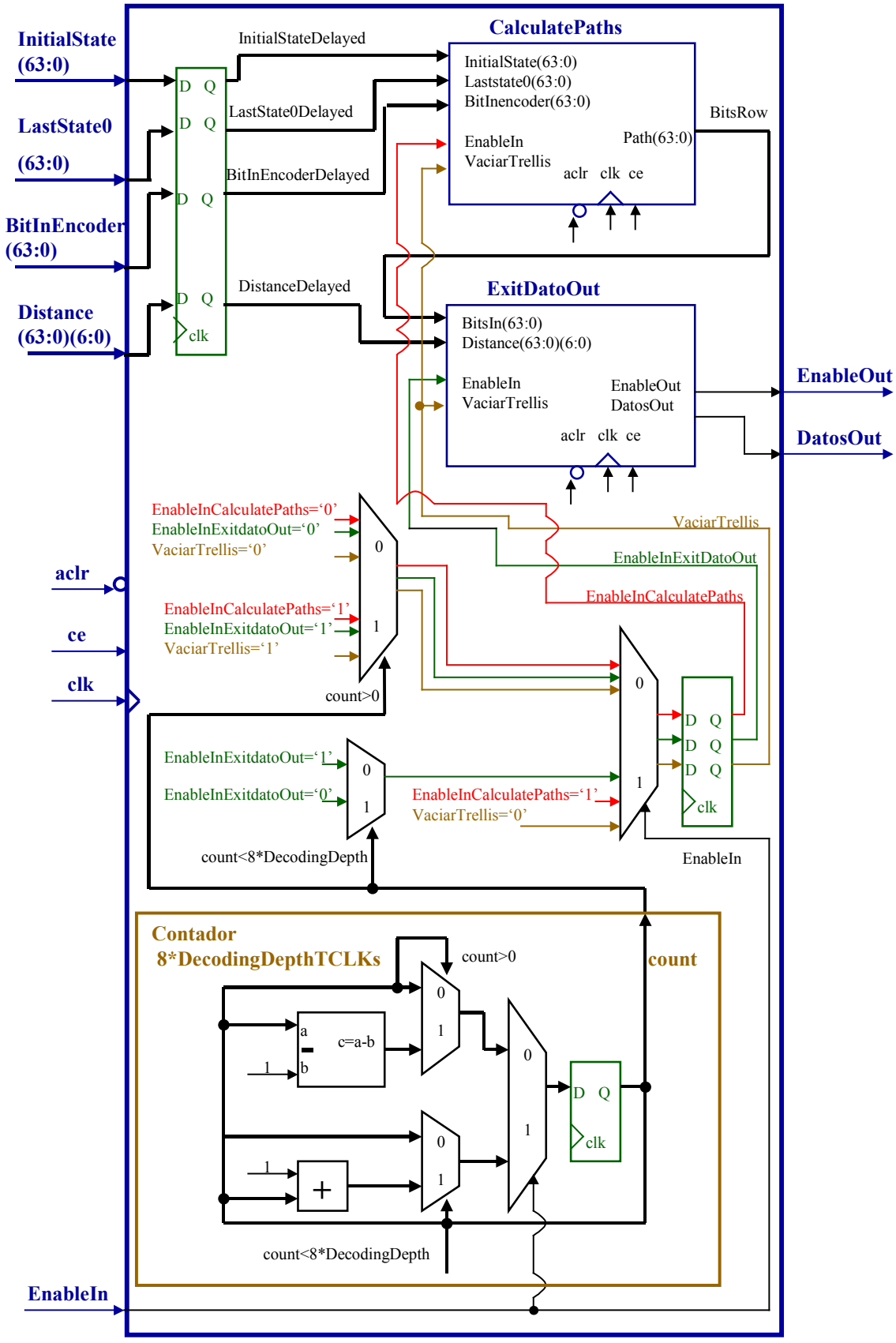


Figura 5.21: RegisterExchange.vhd.

Este es el segundo módulo del decodificador. Trabaja con los datos de salida del ACS y obtiene las dos salidas finales del decodificador, DatosOut y EnableOut. Para implementarlo nos basamos en las referencias del *apartado 2.8.5*.

DatosOut es el bit decodificado final que obtiene el Viterbi a partir de cada dato de entrada compuesto por dos bits. $x = (Din_1,Din_0) \rightarrow DatosOut = F(x)$, un bit.

Este bit final está en la malla trellis. El módulo lo que hace es gestionar la malla actualizándola en cada período de proceso, 8 T_{CLKs}, y buscar en la malla el bit ganador que debe salir en DatosOut en cada período de proceso.

En el trellis en cada período de proceso llega un dato $x = (Din_1,Din_0)$ al ACS. El módulo ACS sitúa ese dato, de manera que en el instante t_x, el dato estará situado en el trellis y se representa con esta información:

- Distance(63:0)(6:0)(t_x).
- LastState0(63:0)(t_x).
- BitInEncoder(63:0)(t_x).
- InitialState(63:0)(t_x).

Las cuatro señales son las entradas del RegisterExchange y contienen toda la información que necesita un decodificador Viterbi para construir el trellis y obtener el bit decodificado ganador.

La decodificación Viterbi consiste en ir almacenando en una memoria los caminos supervivientes hasta cada uno de los estados. Un camino superviviente es la secuencia de bits de entrada al codificador m[n], que hay que seguir para desplazarse en el trellis desde el primer bit, en la posición $t_{x-(DecodingDepth-1)}$ dato $x_{-(DecodingDepth-1)}$ situado. Hasta el último bit, en la posición t_x dato x situado. Por tanto cada uno está formado por decoding depth bits.

Cada camino se representa con una memoria FIFO con (decoding depth) *1 bits. El primer bit estará en la casilla cero y el último en la (decoding depth-1). Entonces necesitamos 64 FIFOs.

Hay que actualizarlos en cada período de proceso:
camino(i)(decoding depth-1:0)(t_x) representa el camino hasta i en t_x.

- camino(i)(0)(t_x) contiene el primer bit del camino i y corresponde al dato que realiza la transición de $t_{x-DecodingDepth}$ a $t_{x-(DecodingDepth-1)}$.

- camino(i)(decoding depth-1)(t_x) contiene el último bit y corresponde al dato que se acaba de escribir en la memoria, el que realiza la transición de t_{x-1} a t_x.

En el siguiente período de proceso los caminos se desplazan, de manera que en camino(i)(decoding depth-1)(t_{x+1}) se escribe el bit que hace pasar de t_x a t_{x+1}. El resto de bits se desplazan una posición, de manera que el que ocupaba la casilla cero sale de la memoria. Pero no basta con eso, el camino hasta i en t_{x+1} no es igual al camino hasta i

en t_x desplazado, sino que será igual al que llega hasta su estado anterior, j ó h en t_x desplazado.

Esto se repite para los 64 estados, por lo que en total sacamos 64 bits de la memoria que forman la señal Path, y entre ellos estará el bit decodificado final que saldrá en DatosOut.

En cada período de proceso se selecciona uno de los caminos supervivientes como el ganador. Será el que llegue hasta el estado i_x que tenga una distancia mínima.

$$\text{Indice camino ganador} = i_x = \min[i \in [0..63]]_{Distance(ix)=\min_{0\le i\le 63}[Distance(i)]=MinDistance}$$

El bit decodificado final es el primer bit del camino superviviente ganador.

Para realizar el trabajo el módulo consta de 2 bloques:

1. CalculatePaths gestiona la memoria que contiene los caminos supervivientes. Los actualiza en cada período de proceso y lee en el puerto Path el primer bit de cada uno de ellos. Ese vector se envía al módulo ExitDatoOut.

2. ExitDatoOut selecciona de entre los 64 caminos cuál es el ganador y saca su primer bit en DatosOut.

Estudio de la latencia de RegisterExchange.

- En t_x el dato x está situado en el trellis y llegan Distance(t_x), LastState0(t_x), BitInEncoder(t_x) e InitialState(t_x).

- El bit correspondiente a decodificar x , DatosOut=F(x) estará disponible cuando hayan transcurrido: (1 T_{CLK} + (decoding depth +1)*8T_{CLKs}). Que es por tanto la latencia del módulo.

- Las entradas al RegisterExchange deben llegar cada 8T_{CLKs}, un período de proceso de un dato. Y permanecer estables durante esos 8 T_{CLKs}.

- En cada período de proceso, hay un nuevo bit decodificado en EnableOut y DatosOut. Estas señales sólo están activas durante un T_{CLK} por período de proceso. Lo hacemos así para que al simular sea más sencillo contar los bits de salida.

- Por tanto en el instante t_x llegan las señales que representan la colocación de x en el trellis, y un T_{CLK} después está en la salida el bit decodificado correspondiente al dato $x_{-DecodingDepth}$.

- El bloque CalculatePaths tiene una latencia de decoding depth períodos de proceso, y el ExitDatoOut de un período de proceso. La latencia extra de 1 T_{CLK} se debe a la generación de las señales de control EnableIn y VaciarTrellis para los dos bloques internos.

Hay tres situaciones diferentes dependiendo de si estamos al inicio de la trama de entrada, momento de inicialización. Cuando ha finalizado la inicialización y se pasa al modo de funcionamiento general, y por último la situación en la que finaliza la trama de entrada. Para regular las tres situaciones tenemos 4 señales de control: EnableIn de entrada al RegisterExchange, EnableInCalculatePaths, EnableInExitDatoOut y VaciarTrellis.

1. En la inicialización, los caminos supervivientes no están completos. Porque cada uno consta de decoding depth bits y entonces hasta que no hayan llegado decoding depth datos al decodificador, no se pueden completar. En esta situación, cada nuevo dato se escribe en las FIFOs, pero no se puede sacar aún ningún dato de la memoria porque los bits válidos no han llegado a la casilla cero. Por tanto en esta situación tendremos:

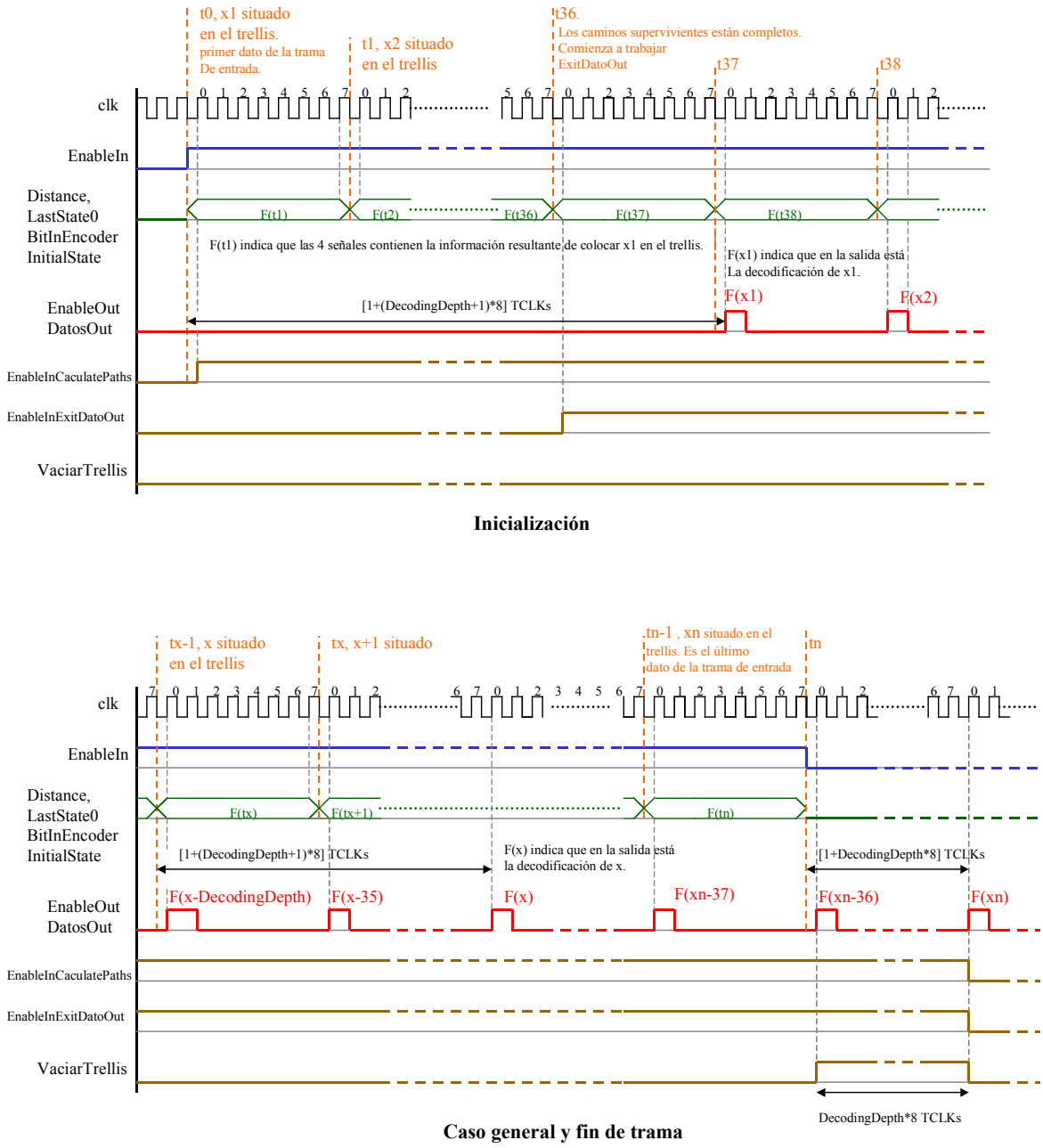
   - EnableIn = ’1’, llegan datos a RegisterExchange.
   - EnableInCalculatePaths = ’1’. El módulo trabaja.
   - EnableInExitDatoOut = EnableOut = DatosOut = ’0’, no hay bits disponibles en la salida.
   - VaciarTrellis = ’0’.

2. Situación general. Una vez que hayan llegado decoding depth datos, las FIFOs ya están llenas. Cuando llega un dato se mete en las FIFOs en la casilla (decoding depth –1), se desplazan los bits una posición y los que ocupaban la casilla cero salen de la memoria y se envían al puerto Path.

   - EnableIn = ’1’, llegan datos a RegisterExchange.
   - EnableInCalculatePaths = ’1’, el módulo trabaja.
   - EnableInExitDatoOut = ’1’ .
   - EnableOut = ‘1’.
   - VaciarTrellis = ’0’.

3. Cuando dejen de llegar datos al módulo, la decodificación aún no habrá terminado. Porque todavía hay que decodificar los decoding depth bits almacenados en la memoria. En esta situación, en cada período de proceso escribimos un cero en la casilla (decoding depth–1) de las FIFOs y desplazamos sus bits una posición, leyendo en Paths los que ocupaban la casilla cero.

   - EnableIn = ’0’, no llegan datos a RegisterExchange.
   - EnableInCalculatePaths = ’1’, el módulo trabaja.
   - EnableInExitDatoOut = ’1’ .
   - EnableOut = ‘1’.
   - VaciarTrellis = ’1’, indica que estamos en la situación 3.



Cronograma 5.7: RegisterExchange.vhd. Decoding depth=36.

Debido a la latencia de un período de proceso del ACS, los instantes de tiempo en RegisterExchange tienen un retardo de 8 T_{CLKs} respecto a esos mismos instantes en ACS.

- En el ACS en t_0 llega el primer dato x_1 al trellis, en t_1 está situado x_1 y llega x_2.
- En el RegisterExchange en t_0 ya está situado el dato x_1 y llega x_2.
- De manera que en el modo genérico en ACS en t_x está situado x y llega x_{+1}. Mientras que en el RegisterExchange en t_x está situado x_{+1} y llega x_{+2}.

5.5.10 RegisterExchange. Funcionamiento con ejemplo de 4 estados.

Para comprender el funcionamiento del módulo, utilizamos un ejemplo sencillo con 4 estados y decoding depth = 5. Tendremos por tanto 4 caminos supervivientes con 5 bits cada uno de ellos. Los representamos así:

camino(i)(4:0)(t_x) Es el camino hasta el estado i en t_x.

- El bit 4 corresponde al último bit. En el trellis es la rama que realiza la transición de t_{x-1} a t_x.

- El bit 0 es el primer bit. Es la rama que realiza la transición de t_{x-5} a t_{x-4}. ($t_{x-DecodingDepth}$ a $t_{x-(DecodingDepth-1)}$).

Trabajamos con el ejemplo del *apartado 2.7*, y nos basamos en la bibliografía citada en *2.8.2*.

El RegisterExchange que hemos implementado consta de 64 estados y decoding depth de 36 bits y de 8 bits, porque hemos realizado dos diseños completos. Su funcionamiento es equivalente al ejemplo de 4 estados, con la diferencia de que hay que manejar 64 FIFOs de 36 ó de 8 bits.

Caso General. Situación 2.

Nota 5.4: Este caso ya lo tratamos en el *apartado 5.4.3*. Aquí ampliamos un poco la explicación.

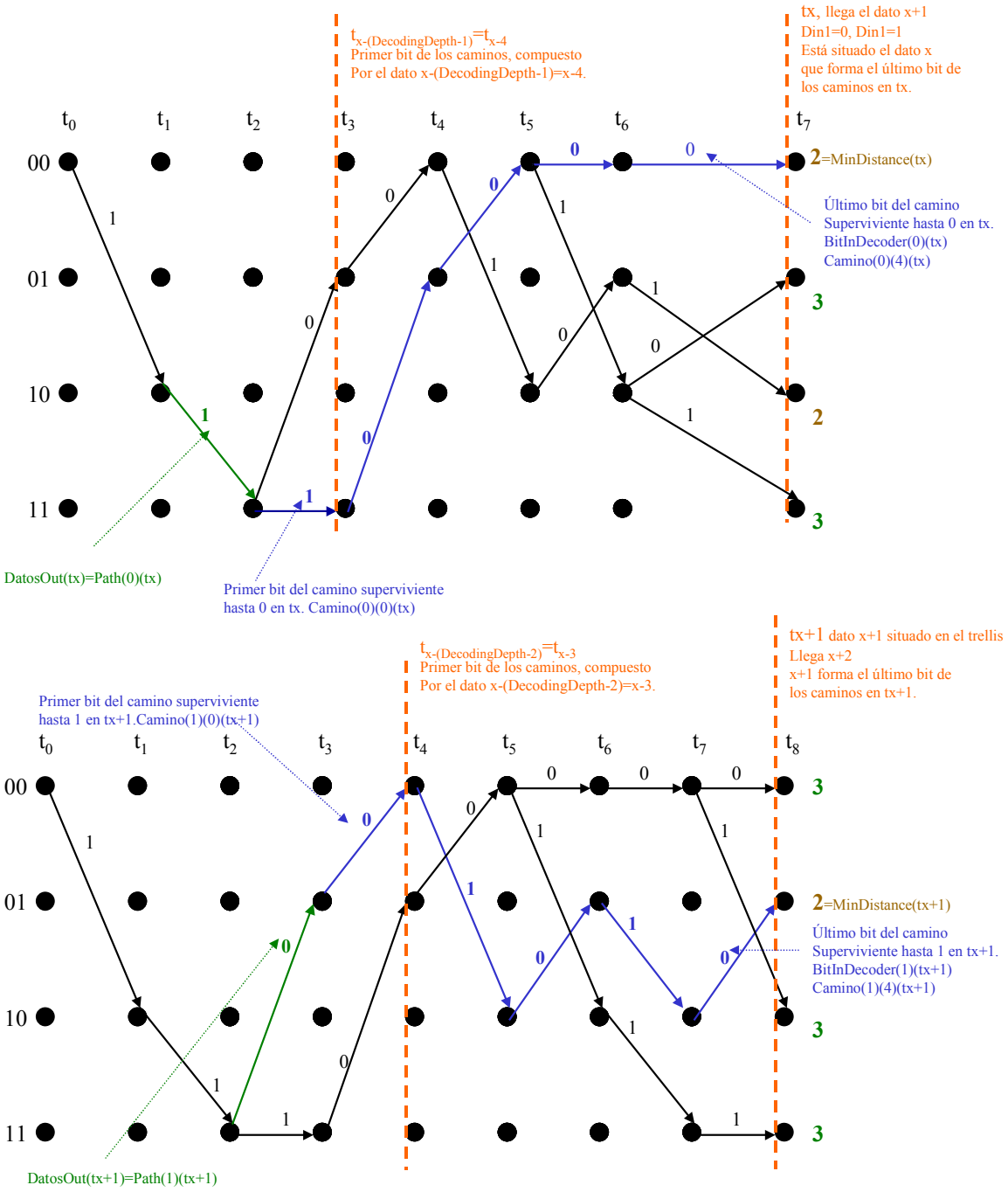


Figura 5.22: Trellis caso general.

Tabla 5.10: Información contenida en el trellis							
t_x				t_{x+1}			
Camino(0)(4:0)	**00001**			Camino(0)(4:0)	00000		
Camino(1) (4:0)	01001			**Camino(1)(4:0)**	**01010**		
Camino(2) (4:0)	10100			Camino(2)(4:0)	10000		
Camino(3) (4:0)	11001			Camino(3)(4:0)	11100		
Path(3:0)	1111	DatosOut	**'1'**	Path(3:0)	1101	DatosOut	**'0'**
Distance(3:0)	(3,2,3,**2**)	BitInEncoder(3:0)	1100	Distance(3:0)	(3,3,**2**,3)	BitInEncoder(3:0)	1100
LastState0(3:0)	1000	InitialState(3:0)	1111	LastState0(3:0)	1000	InitialState(3:0)	1111

Cuando comienza el período de proceso que transcurre entre t_x y t_x+7 T_{CLKs} la memoria contiene los caminos supervivientes correspondientes a t_x. El ganador puede ser el dos o el cero, porque:

MinDistance(t_x) = 2 = Distance(2)(t_x) = Distance(0)(t_x).
Hemos elegido aleatoriamente el cero como el ganador, así que el camino superviviente ganador es camino(0)(4:0)(t_x) = "**00001**" y **DatosOut(t_x) = '1' = Path(0)(t_x)**.

Según va transcurriendo el período de proceso RegisterExchange va actualizando los caminos. De tal manera que al llegar t_{x+1}, los nuevos caminos ya están almacenados en la memoria. En t_{x+1} tenemos MinDistance(t_{x+1}) = 2 =Distance(1)(t_{x+1}), entonces el ganador es camino(1)(t_{x+1})="**01010**" y **DatosOut(t_{x+1}) = '0' = Path(1)(t_{x+1})**.

En este ejemplo explicamos cómo el módulo actualiza los caminos cuando el tiempo transcurre entre t_x y t_x + 7T_{CLKs}. De esta forma al llegar el siguiente período de proceso t_{x+1}= t_x + 8 T_{CLKs} la actualización ya estará guardada en la memoria. El funcionamiento es el siguiente:

Datos de partida:
Las entradas al módulo permanecerán constantes durante todo el período de proceso, de t_x a t_x + 7 T_{CLKs}. Cuando llegue el instante t_{x+1} las entradas serán diferentes.

Tabla 5.11: Entradas en RegisterExchange.	
Distance(3:0)(t_{x+1})	(3,3,**2**,3)
BitInEncoder(3:0) (t_{x+1})	('1','1','0','0')
LastState0(3:0) (t_{x+1})	('1','0','0','0')
InitialState(3:0) (t_{x+1})	('1','1','1','1')

Los pasos a seguir son:

1. El último bit del camino hasta i en t_{x+1} es igual a BitInEncoder(i)(t_{x+1}). Camino(i)(decoding depth-1)(t_{x+1})= BitInEncoder(i)(t_{x+1}).

2. Al escribir un nuevo bit hay que desplazar una posición los que ya están, por tanto el que ocupaba la casilla cero, camino(i)(0)(t_x) sale de la memoria. Es una estructura FIFO.

3. Los bits (decoding depth-2)..0 del camino hasta i en t_{x+1} serán los bits (decoding depth-1)..1 del camino hasta el estado anterior j ó h, en t_x. Esta circunstancia es lo que da más complejidad al decodificador, porque de un período de proceso al siguiente cambian todos los bits de la memoria.

Siguiendo los pasos 1 2 y 3 obtenemos:

- Camino(i)(t_{x+1})= BitInEncoder(i)(t_{x+1})&camino(j)(4:1)(t_x); Si el estado anterior al i fuese el j. LastState0(t_{x+1}) = '1'

.

- Camino(i)(t_{x+1})= BitInEncoder(i)(t_{x+1})&camino(h)(4:1)(t_x); Si el estado anterior al i fuese el h. LastState0(t_{x+1})='0'.

4. Al finalizar el paso 3, la memoria ya está actualizada. Entonces lo que hay que hacer es enviar a Path los bits que han salido de las FIFOs a consecuencia de haber escrito un nuevo bit. Esos bits son los que ocupaban la primera posición de cada uno de los caminos en t_x. Concretamente:

   - Path(i)(t_{x+1})= camino(j)(0)(t_x); Si el estado anterior al i fuese el j.
   - Path(i)(t_{x+1})= camino(h)(0)(t_x); Si el estado anterior al i fuese el h.

Aplicándolo al ejemplo:

- LastState0(t_{x+1})="1000" así que:
  - El estado anterior al 3 es el j = 3.
  - El estado anterior al 2 es el h = 0.
  - El estado anterior al 1 es el h = 2.
  - El estado anterior al 0 es el h = 0.

Tabla 5.12: Contenido de la memoria al llegar t_{x+1}.

Caminos supervivientes		Señal Path
4→último bit 0→primer bit	t_{x+1} 43210	t_{x+1}
Camino(0)	00000=BitIn_I_Encoder(0)(t_{x+1})&Camino(0)(4:1)(t_x)	Path(0)='1'=Camino(0)(0)(t_x)
Camino(1)	**01010**=BitIn_I_Encoder(1) (t_{x+1})&Camino(2)(4:1)(t_x)	**Path(1)='0'**=Camino(2)(0)(t_x)
Camino(2)	10000=BitIn_I_Encoder(2) (t_{x+1})&Camino(0)(4:1)(t_x)	Path(2)='1'=Camino(0)(0)(t_x)
Camino(3)	11100=BitIn_I_Encoder(3) (t_{x+1})&Camino(3)(4:1)(t_x)	Path(3)='1'=Camino(3)(0)(t_x)

Los pasos 1, 2, 3 y 4 los realiza el bloque CalculatePaths.

ExitDatoOut recibe la señal Path y selecciona entre ellos el bit decodificado final. Será el primer bit del camino que tenga una distancia mínima en t_{x+1}. En el ejemplo MinDistance(t_{x+1})= 2 = Distance(1), así que el camino ganador es el uno, y por tanto: **DatosOut(t_{x+1}) = Path(1)(t_{x+1})='0'.**

Inicialización, situación 1.

El proceso es el mismo que en la situación 2. Las diferencias son que como aún no han llegado bits válidos a la casilla cero de las FIFOS, en el vector Path sólo hay ceros. Y que ExitDatoOut no trabaja porque no está habilitado.

Fin de la trama de entrada. Situación 3.

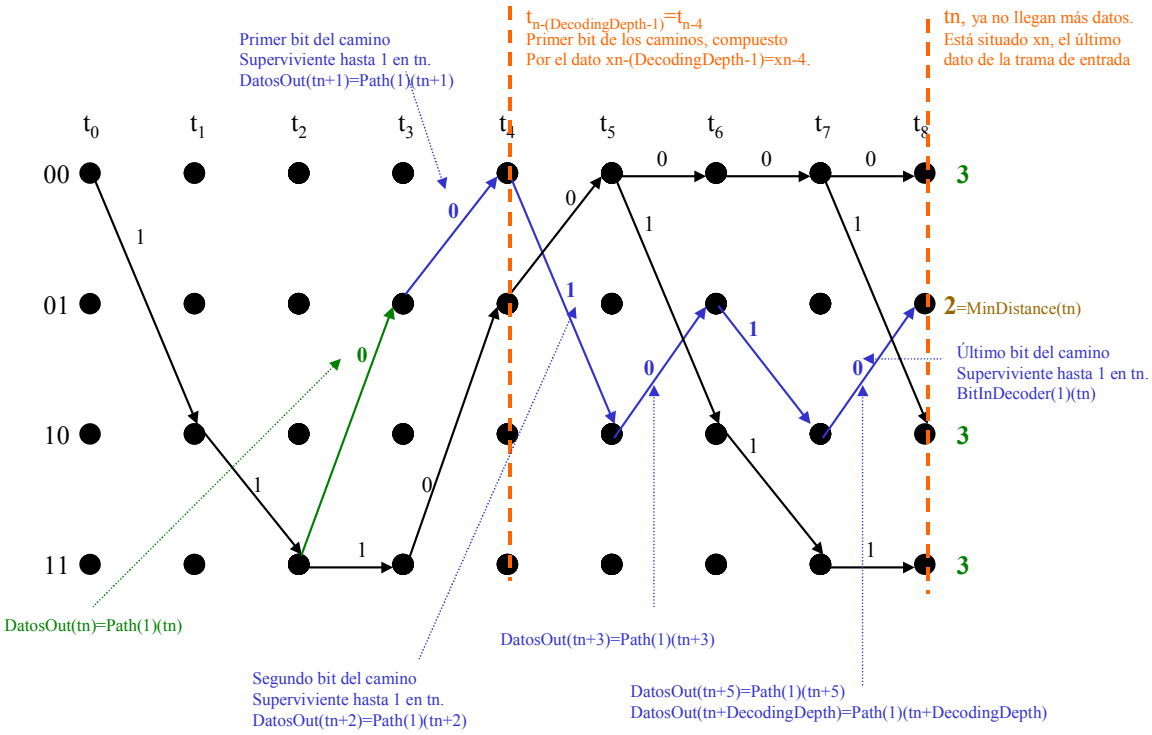


Figura 5.23: Trellis final de la trama.

Tabla 5.13: Información contenida en el trellis.						
	tn	tn+1	tn+2	tn+3	tn+4	tn+5
Camino(0)(4:0)	00000	00000	00000	00000	00000	00000
Camino(1)(4:0)	**01010**	00101	00010	00001	00000	00000
Camino(2)(4:0)	10000	01000	00100	00010	00001	00000
Camino(3)(4:0)	11100	01110	00111	00011	00001	00001
Path(3:0)	1101	0000	0010	1000	1010	1100
DatosOut	**'0'**	**'0'**	**'1'**	**'0'**	**'1'**	**'0'**
Distance(3:0)	(3,3,2,3)					
BitInEncoder(3:0)	1100					
Laststate0	1000					
InitialState(3:0)	1111					

En t_n el último dato de la trama de entrada x_n está situado en el trellis.

Cuando comienza el período de proceso que transcurre entre t_n y t_n+7 T_{CLKs} la memoria contiene los caminos supervivientes correspondientes a t_n. Tenemos: MinDistance(t_n)=2=Distance(1)(t_n), entonces el ganador es camino(1)(4:0)(t_n)="01010" y **DatosOut(t_n) = '0' = Path(1)(t_n).**

En los siguientes períodos de proceso ya no llegan más datos al módulo, pero la decodificación aún no ha terminado, porque hay que sacar los bits almacenados en la memoria.

Los últimos bits decodificados por el Viterbi corresponden a los del último camino superviviente ganador en t_n, de manera que en t_{n+1} en DatosOut debemos leer el primer bit del camino ganador y en $t_{n+DecodingDepth}$ el último bit. En el ejemplo tendremos.

1. DatosOut(t_{n+1}) = camino(1)(0)(t_n) = **'0'**.
2. DatosOut(t_{n+2}) = camino(1)(1)(t_n) = **'1'**.
3. DatosOut(t_{n+3}) = camino(1)(2)(t_n) = **'0'**.
4. DatosOut(t_{n+4}) = camino(1)(3)(t_n) = **'1'**.
5. DatosOut(t_{n+5}) = camino(1)(4)(t_n) = **'0'**.

A continuación explicamos como se realiza la actualización entre t_n y t_n + 7 T_{CLKs}. El funcionamiento en los siguientes períodos es el mismo.

Las entradas al módulo permanecerán constantes entre t_n y $t_{n+DecodingDepth}$, pero RegisterExchange sólo lee Distance(t_n), el resto de entradas no se utilizan en esta situación.

Escribimos un cero en la casilla (decoding depth-1) de cada uno de los caminos y desplazamos una posición el resto de bits, de manera que el que ocupaba la casilla cero sale de la FIFO:

1. El último bit del camino hasta i en t_{n+1} es igual a '0'. Camino(i)(4)(t_{n+1})= '0'.

2. Al escribir un nuevo bit hay que desplazar una posición los que ya están, por tanto el que ocupaba la casilla cero, camino(i)(0)(t_n) sale de la memoria. Es una estructura FIFO.

3. Los bits (decoding depth-2)..0 del camino hasta i en t_{n+1} serán los bits (decoding depth-1)..1 del camino hasta i en t_n.

Siguiendo los pasos 1 2 y 3 obtenemos:

4. Camino(i)(t_{n+1})= '0'&camino(i)(4:1)(t_n).

5. Al finalizar el paso 3, la memoria ya está actualizada. Entonces lo que hay que hacer es enviar a Path los bits que han salido de las FIFOs a consecuencia de haber escrito un nuevo bit. Esos bits son los que ocupaban la primera posición de cada uno de los caminos en t_n. Concretamente: Path(i)(t_{n+1})= camino(i)(0)(t_n);

Los pasos 1, 2, 3 y 4 los realiza el bloque CalculatePaths.

ExitDatoOut recibe la señal Path y selecciona entre ellos el bit decodificado final. En esta situación el camino ganador es siempre el mismo en todos los períodos de proceso, el que tenga MinDistance(t_n). En el ejemplo MinDistance(t_n)= 2 = Distance(1), así que el ganador es el uno, y por tanto: DatosOut(t_{n+1}) = Path(1)(t_{n+1})='0'.

5.5.11 CalculatePaths.vhd.

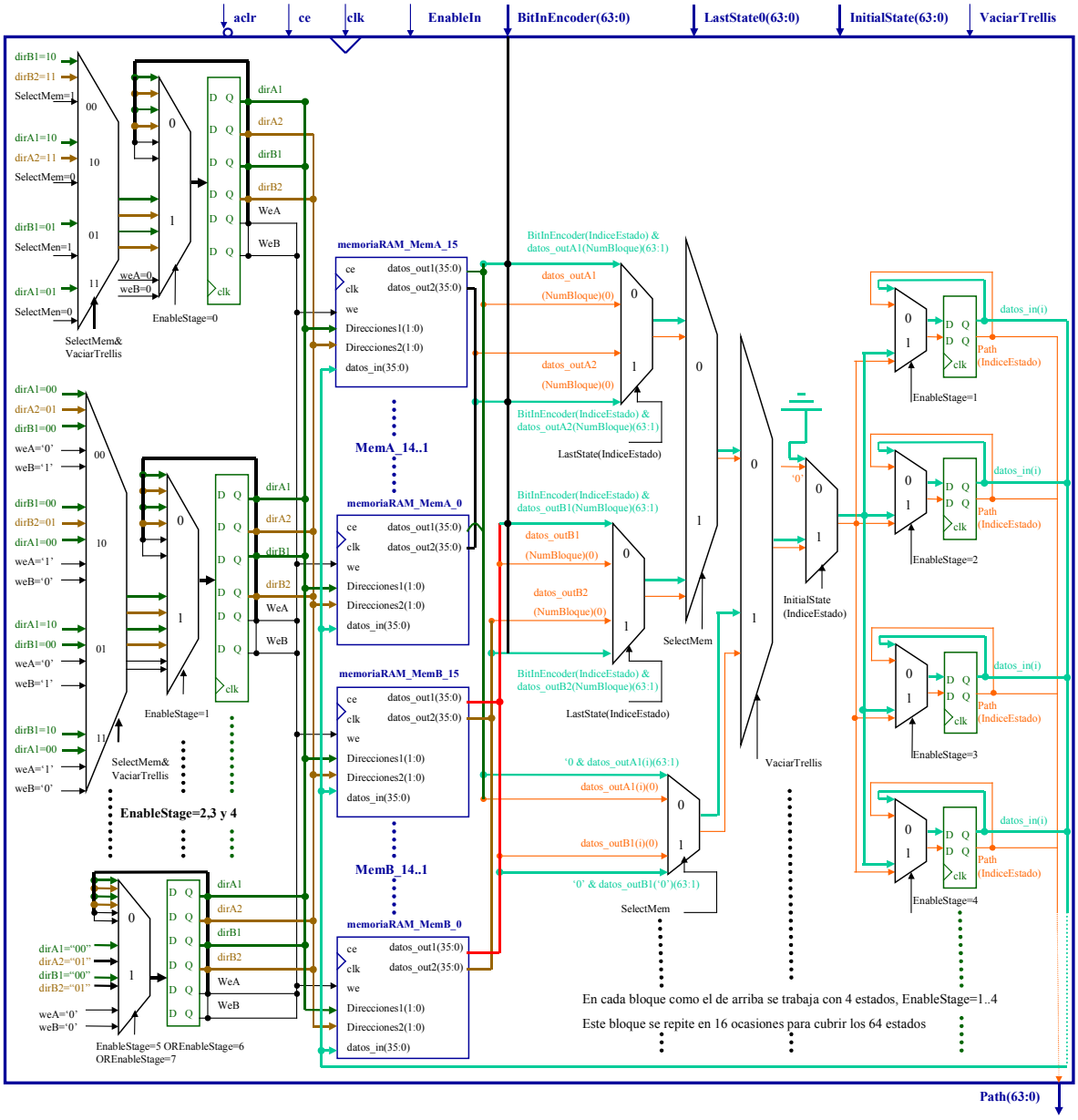


Figura 5.24: CalculatePats.vhd

Gestiona la memoria que contiene los caminos supervivientes. Los actualiza en cada período de proceso y lee en el puerto Path el primer bit de cada uno de ellos. Ese vector se envía al módulo ExitDatoOut.

La función de este módulo en la decodificación Viterbi la hemos desarrollado en los dos apartados anteriores. En este solamente explicaremos la estructura interna.

Cada camino se escribe en una memoria FIFO con (decoding depth) *1 bits.
camino(i)(decoding depth-1:0)(t_x) representa el camino hasta i en t_x. La casilla (decoding depth-1) corresponde al último bit y la cero al primero.

Hay tres modos de trabajo, que son iguales que en RegisterExchange:

1. La inicialización en la que los caminos supervivientes no están completos. Este modo está determinado por EnableIn = ’1’ y VaciarTrellis = ’0’. Tiene la particularidad de que en Path sólo hay ceros porque los bits escritos en las FIFOs aún no han llegado a la casilla cero. En cada período de proceso hacemos:

   - Camino(i)(t_{x+1})= BitInEncoder(i)(t_{x+1})&camino(j)(4:1)(t_x); Si el estado anterior al i fuese el j. LastState0(t_{x+1}) = ’1’

.

   - Camino(i)(t_{x+1})= BitInEncoder(i)(t_{x+1})&camino(h)(4:1)(t_x); Si el estado anterior al i fuese el h. LastState0(t_{x+1})=’0’.

   - Path(i)(t_{x+1})= camino(j)(0)(t_x); Si el estado anterior al i fuese el j.
   - Path(i)(t_{x+1})= camino(h)(0)(t_x); Si el estado anterior al i fuese el h.

2. Modo general, ocurre cuando ya han llegado decoding depth datos al módulo. El funcionamiento es el mismo que en el modo 1, con la diferencia de que en Path hay bits válidos. También está determinado por EnableIn = ‘1’ y VaciarTrellis=‘0’.

3. Fin de la trama de entrada, se caracteriza por por EnableIn = ‘1’ y VaciarTrellis=‘1’. En cada período de proceso hacemos:

   - Camino(i)(t_{x+1})= ‘0’&camino(i)(4:1)(t_x).
   - Path(i)(t_{x+1})= camino(i)(0)(t_x).

La forma más sencilla e inmediata de definir la estructura del módulo sería mediante registros. Consistiría en almacenar cada camino en un vector con decoding depth registros. Tendríamos un módulo con 64*(decoding depth) registros que iríamos actualizando en cada período de proceso. Hemos implementado este diseño en VHDL y funciona correctamente. Pero para nuestra aplicación particular no es viable porque ocupa demasiada área. Solamente este módulo CalculatePaths ocupa un 28% de capacidad de la FPGA que usamos en el diseño: Virtex 4 XC4VSX55 10FF1148. En el *apartado 7.2.4* y la *tabla 7.3* explicamos esta implementación.

Entonces para poder implementar el diseño en la FPGA no podemos realizar este módulo con registros. Lo realizamos con memoria RAM y de esta forma conseguimos reducir el área de CalculatePaths a un 9% de la capacidad de la FPGA, ver de nuevo *apartado 7.2.4* y la *tabla 7.3*

Nuestro diseño optimiza el área y no penaliza la velocidad, porque el módulo crítico que limita la frecuencia máxima del decodificador Viterbi es FindMinIndex. Tiene el único inconveniente de que la estructura del código es muy compleja. Como veremos a continuación la comprensión del código no es nada intuitiva.

Estructura de CalculatePaths.

En el período de proceso que transcurre entre t_x y t_x+7T_{CLKs} se leen los caminos en el instante t_x y se van escribiendo los actualizados correspondientes a t_{x+1}. De tal manera que en el siguiente período que transcurre entre t_{x+1} y $t_{x+1}+7$ T_{CLKs}, se leen los caminos correspondientes a t_{x+1} y se escriben los correspondientes a t_{x+2}.

Esta característica obliga a tener dos memorias RAM, cada una con 64*(decoding depth bits). Las llamamos MemA y MemB y se van alternando, de manera que entre t_x y t_x+7T_{CLKs} los caminos correspondientes a t_x están en MemA. Por tanto se leen de MemA y los actualizados correspondientes a t_{x+1} se escriben en MemB. En el siguiente período se invierte el orden, leemos de MemB y escribimos en MemA.

Las memorias RAM las realizamos íntegramente en VHDL, sin usar ningún elemento dependiente de la tecnología. Se trata de memorias con doble puerto de lectura síncrona y un puerto de escritura síncrona. Como veremos en el siguiente apartado esta opción es la más óptima para el diseño. El inconveniente es que dificulta mucho la realización del código.

En los modos de trabajo 1 y 2 para escribir el camino hasta i en t_{x+1} necesitamos leer los de j y h en t_x. Por tanto en cada período de proceso para actualizar los 64 caminos necesitamos realizar 64 escrituras y 128 lecturas.

Como las memorias son síncronas con un puerto de escritura y 2 de lectura, sólo podemos realizar una escritura y dos lecturas en cada T_{CLK}. Este módulo tiene un período de proceso de 8 T_{CLKs}, por tanto para poder realizar las 64 escrituras tenemos que dividir la memoria en varios módulos.

Las RAMs MemA y MemB se dividen cada una de ellas en 16 módulos de 4*(decoding depth) bits cada uno. Cada módulo constituye una memoria RAM independiente con dos puertos de lectura y uno de escritura. De esta manera en cada T_{CLK} se pueden realizar 32 lecturas y 16 escrituras en paralelo. Entonces para cubrir los 64 estados dividimos la actualización en cuatro pasos, actualizando 16 caminos en cada uno de ellos.

0. Paso cero, actualizamos los caminos hasta los estados i = 0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60.

1. Actualizamos los caminos hasta los estados i = 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61.

2. Paso 2, i = 2, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62.

3. Paso 3, i = 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63.

MemA y MemB se organizan de manera que el módulo cero almacena los caminos hasta los estados 3..0, con este direccionamiento:

0. "00" estado 0.
1. "01" estado 1.
2. "10" estado 2.
3. "11" estado 3.

El orden ascendente es el mismo en el resto de módulos, de manera que el 15 almacena los caminos hasta los estados 63..60. 63 en la dirección "11" y 60 en la "00".

La organización de las señales de la memoria es la siguiente:

En cada paso se actualizan 16 estados en paralelo, así que las señales de direccionamiento y datos de las memorias son vectores con 16 señales cada una.

Si toca leer de MemA y escribir en MemB.

- dirA1: Si VaciarTrellis = '0' → Dirección del camino hasta h en t_x. Si VaciarTrellis = '1'→ Dirección del camino hasta i en t_x.
- dirA2: Dirección del camino hasta j en t_x.
- DirB1: Dirección del camino hasta i en t_{x+1}.
- datos_outA1: Si VaciarTrellis='0' → Lectura del camino hasta h en t_x. Si VaciarTrellis = '1' → Lectura del camino hasta i en t_x.
- datos_outA2: Lectura del camino hasta j en t_x.
- datos_in: puerto de escritura de MemB. Para escribir el camino hasta i en t_{x+1}.

Si toca leer de MemB y escribir en MemA.

- DirB1: Si VaciarTrellis = '0' → Dirección del camino hasta h en t_x. Si VaciarTrellis = '1' → Dirección del camino hasta i en t_x.
- DirB2: Dirección del camino hasta j en t_x.
- DirA1: Dirección del camino hasta i en t_{x+1}.
- datos_outB1: Si VaciarTrellis ='0' → Lectura del camino hasta h en t_x. Si VaciarTrellis = '1' → Lectura del camino hasta i en t_x.
- datos_outB2: Lectura del camino hasta j en t_x.
- datos_in: puerto de escritura de MemA. Para escribir el camino hasta i en t_{x+1}.

Proceso de actualización de los caminos y de Path entre t_x y t_x+7T_{CLKs}.

(Ver *cronogramas 5.8* y *5.9*)

Suponemos el caso en el que se lee de MemA y se escribe en MemB. El caso contrario es equivalente, sólo hay que cambiar dirA por dirB y datos_outA por datos_outB.

En t_x + $5T_{CLKs}$ los 64 bits de Path están actualizados, contendrán los valores correspondientes a Path(t_{x+1}). Por tanto la latencia de CalculatePaths es de 5 T_{CLKs}.

La actualización se realiza en 4 pasos y en cada uno de ellos hay que realizar las 4 siguientes actividades. Explicamos el proceso para actualizar únicamente un estado: camino(i)(t_{x+1}) y Path(i)(t_{x+1}). Como en cada paso se actualizan 16 estados, las actividades 0..3 se repiten 16 veces en paralelo.

0. En t_x, en dirA1 y dirA2 están disponibles las direcciones de lectura. Si VaciarTrellis='0' serán las de los caminos hasta los estados j y h en t_x anteriores a i. Si VaciarTrellis='1' entonces sólo interesa dirA1 que contendrá la dirección para leer el camino hasta i en t_x.

1. en t_x + $1T_{CLK}$ en datos_outA1 y datos_outA2 estarán las lecturas correspondientes a las casillas indicadas por dirA1 y dirA2 en el anterior T_{CLK}. Así que tendremos:
   - datos_outA1(t_x+1T_{CLK})=camino(h)(t_x), si VaciarTrellis = '0'.
   - datos_outA2(t_x+1T_{CLK})=camino(j)(t_x), si VaciarTrellis = '0'.
   - datos_outA1(t_x+1T_{CLK})=camino(i)(t_x), si VaciarTrellis = '1'.

   En datos_outA1 y A2 y las entradas BitInEncoder(t_{x+1}) y LastState0(t_{x+1}), tenemos la información necesaria para calcular el camino hasta i en t_{x+1},y el bit Path(i) en t_{x+1}. Los resultados se los asignamos a datos_in y al puerto Path.
   Además en dirB1 asignamos la dirección para escribir en la fila i en MemB.

2. En t_x+ $2T_{CLKs}$, en Path(i) ya está el valor actualizado, correspondiente a t_{x+1}.Y en dirB1 y datos_in se leen los valores asignados en el T_{CLK} anterior.

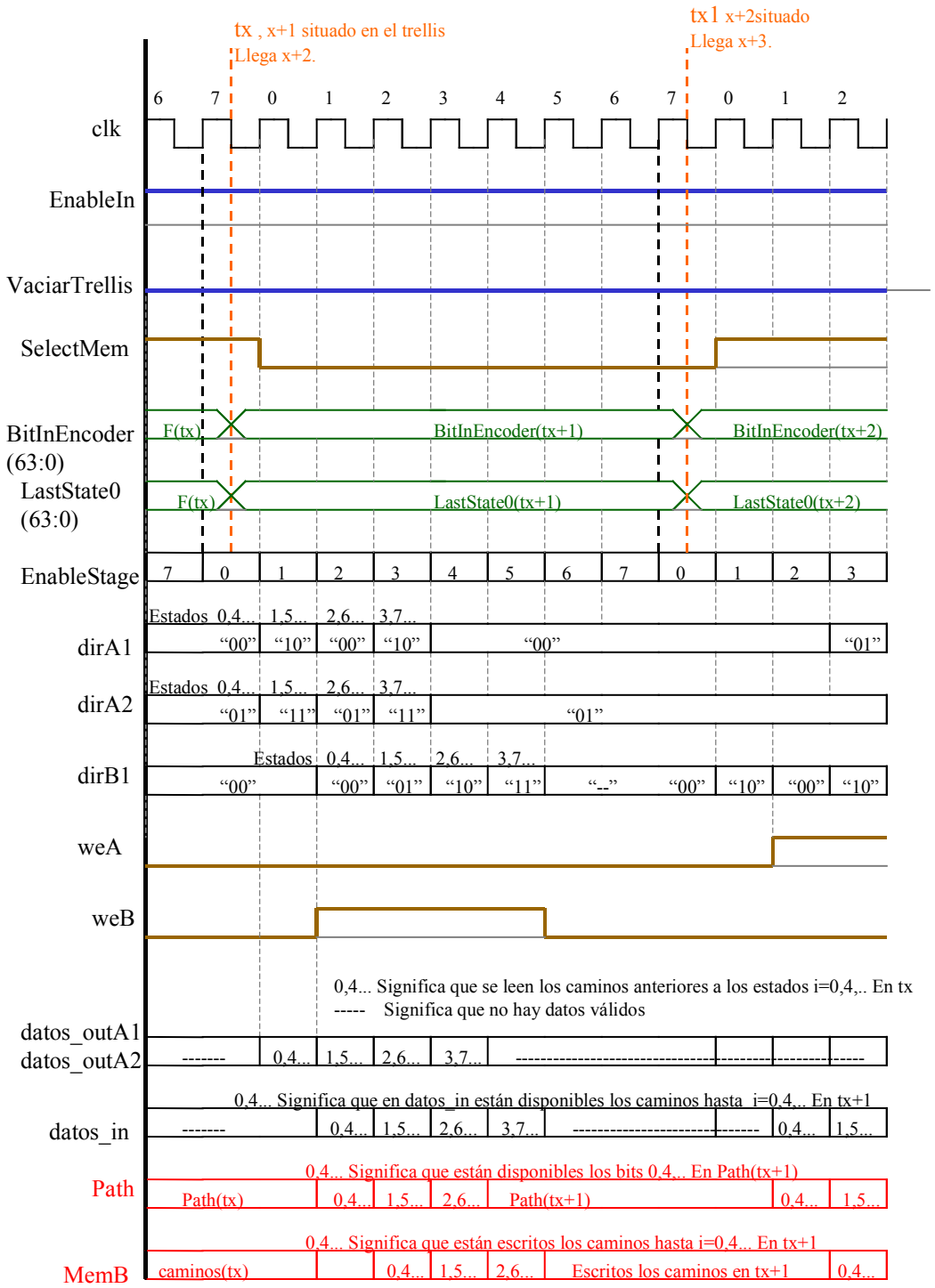
3. En t_x + $3T_{CLKs}$ ya está escrito en MemB el camino hasta i en t_{x+1}.

Los pasos se van solapando, de manera que en t_x comienza el cálculo de los caminos anteriores a i= 0, 4, 8 ....; En t_x+2T_{CLKs} finaliza la actualización de Path(i)(t_{x+1}) y en t_x+3T_{CLKs} finaliza la escritura de los i caminos en MemB.

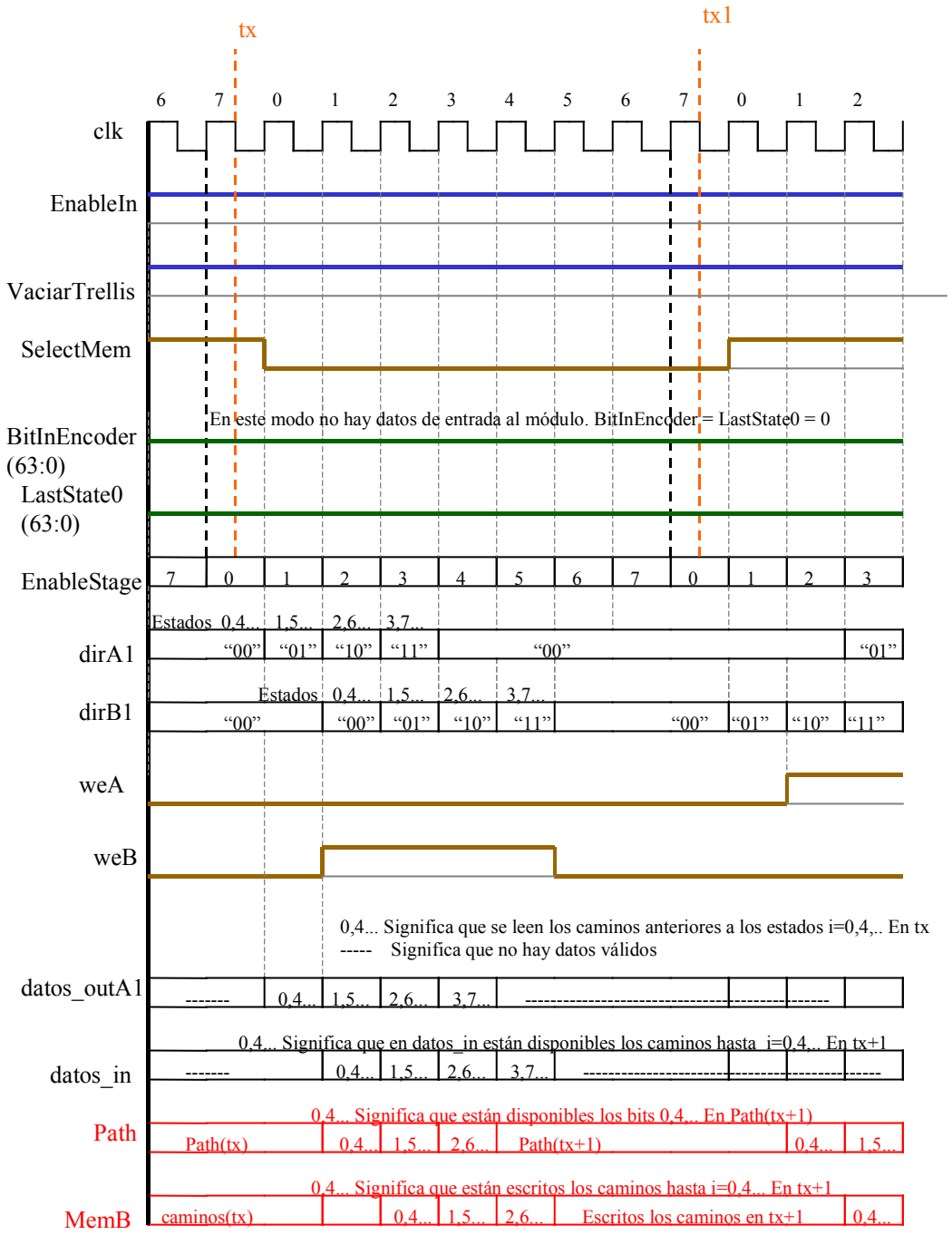
En t_x+1T_{CLK} comienza el cálculo de los caminos anteriores a i = 1, 5, 9 ....; De manera que en t_x+3T_{CLKs} estará actualizado Path(i) y en t_x+4T_{CLKs} estará escrita MemB.

En t_x+2T_{CLKs} comenzamos con i = 2, 6, 10 ....; En t_x+4T_{CLKs} estará actualizado Path(i) y en t_x+5T_{CLKs} estará escrita MemB.

En t_x+3T_{CLKs} comenzamos con i=3, 7, 11....; Entonces en t_x+5T_{CLKs} en Path(t_{x+1}) estarán los 64 bits actualizados, y en t_x+6T_{CLKs} MemB contendrá los 64 caminos hasta i en t_{x+1}.



Cronograma 5.8: CalculatePaths.vhd. Modo General.
Lectura de MemA, escritura en MemB.



Cronograma 5.9: CalculatePaths.vhd. Fin de la trama de entrada.
Lectura de MemA, escritura en MemB.

Direccionamiento de la memoria.

Tabla 5.14: Organización de la memoria RAM.

Módulo de j y h	EstadoAnterior Dirección	Estado i Dirección	Módulo de i	Módulo de j y h	EstadoAnterior Dirección	Estado i Dirección	Módulo de i
15	J: 63, “11”	63, “11”	15	15	j: 63, “11”	31, “11”	7
	H: 62, “10”				h: 62, “10”		
	J: 61, “01”	62, “10”			j: 61, “01”	30, “10”	
	H: 60, “00”				h: 60, “00”		
14	j: 59	61, “01”		14	j: 59	29, 01”	
	h: 58				h: 58		
	j: 57	60, “00”			j: 57	28, “00”	
	h: 56				h: 56		
13	j: 55	59	14	13	j: 55	27	6
	h: 54				h: 54		
	j: 53	58			j: 53	26	
	h: 52				h: 52		
12	j: 51	57		12	j: 51	25	
	h: 50				h: 50		
	j: 49	56			j: 49	24	
	h: 48				h: 48		
11	j: 47	55	13	11	j: 47	23	5
	h: 46				h: 46		
	j: 45	54			j: 45	22	
	h: 44				h: 44		
10	j: 43	53		10	j: 43	21	
	h: 42				h: 42		
	j: 41	52			j: 41	20	
	h: 40				h: 40		
9	j: 39	51	12	9	j: 39	19	4
	h: 38				h: 38		
	j: 37	50			j: 37	18	
	h: 36				h: 36		
8	j: 35	49		8	j: 35	17	
	h: 34				h: 34		
	j: 33	48			j: 33	16	
	h: 32				h: 32		
7	j: 31	47	11	7	j: 31	15	3
	h: 30				h: 30		
	j: 29	46			j: 29	14	
	h: 28				h: 28		
6	j: 27	45		6	j: 27	13	
	h: 26				h: 26		
	j: 25	44			j: 25	12	
	h: 24				h: 24		
5	j: 23	43	10	5	j: 23	11	2
	h: 22				h: 22		
	j: 21	42			j: 21	10	
	h: 20				h: 20		
4	j: 19	41		4	j: 19	9	
	h: 18				h: 18		
	j: 17	40			j: 17	8	
	h: 16				h: 16		
3	j: 15	39	9	3	j: 15	7	1
	h: 14				h: 14		
	j: 13	38			j: 13	6	
	h: 12				H: 12		
2	j: 11	37		2	J: 11	5	
	h: 10				H: 10		
	j: 9	36			j: 9	4	
	h: 8				h: 8		
1	j: 7	35, “11”	8	1	j: 7	3, “11”	0
	h: 6				h: 6		
	j: 5	34, “10”			j: 5	2, 10”	
	h: 4				h: 4		
0	j: 3, “11”	33, “01”		0	j: 3, “11”	1, “01”	
	H: 2, “10”				h: 2, “10”		
	J: 1, “01”	32, “00”			j: 1, “01”	0, “00”	
	H: 0, “00”				h: 0, “00”		

Debemos determinar en que módulo de los 16 que forman cada memoria debemos leer y escribir. Y a continuación seleccionar una de sus 4 casillas. Suponemos el caso en el que se lee de MemA y se escribe en MemB.

Para determinar el número de módulo definimos la variable indice = parte entera [i/4]. En los 4 pasos el módulo de MemB siempre será el que indica índice.

En el primer paso se actualizan los caminos hasta los estados i = 0, 4, 8 ... Entonces DirB1= “00”. Ejemplo si i=44 escribimos en la casilla cero del módulo 11 de MemB.
Si VaciarTrellis = '0' Para cada estado i, su anterior estará en:

- DirA1="00" , DirA2="01". Y de entre los 16 bloques de MemA en el que cumpla:
    - NumBloque = 2*indice; si indice < 8.
    - NumBloque = 2*(indice-8); si indice >= 8.

Ejemplo si i=8 → sus estados anteriores son el j=17 y el h=16 y se encuentran en el módulo 4 de MemA, en las casillas “01” y “00”.

Si VaciarTrellis='1' → DirA1="00". El número de módulo en MemA es igual a indice.

En el segundo paso se actualizan los caminos hasta i = 1, 5, 9... → DirB1= “01”.
Si VaciarTrellis = '0' Para cada estado i, su anterior estará en:

- DirA1="10" , DirA2="11". Y el bloque de MemA será:
    - NumBloque = 2*indice; si indice < 8.
    - NumBloque:= 2*(indice-8); si indice >= 8.

Ejemplo si i = 45 → sus estados anteriores son el j=27 y el h=26 y se encuentran en el módulo 6 de MemA, en las casillas “10” y “11”.

Si VaciarTrellis='1' → DirA1="01". El número de módulo en MemA es igual a indice.

En el tercer paso se actualizan los caminos hasta i = 2, 6, 10 ... → DirB1= “10”.
Si VaciarTrellis = '0' Para cada estado i, su anterior estará en:

- DirA1="00" , DirA2="01". Y el bloque de MemA será:
    - NumBloque = 2*indice + 1 si indice < 8.
    - NumBloque:= 2*(indice-8) + 1 si indice >= 8.

Si VaciarTrellis='1' → DirA1="10". El número de módulo en MemA es igual a indice.

En el cuarto paso se actualizan los caminos hasta i = 3, 7, 11 ... →DirB1= “11”.
Si VaciarTrellis = '0' Para cada estado i, su anterior estará en:

- DirA1="10" , DirA2="11". Y el bloque de MemA será:
    - NumBloque = 2*indice + 1 si indice < 8.
    - NumBloque = 2*(indice-8) + 1 si indice >= 8.

Si VaciarTrellis='1' → DirA1="11". El número de módulo en MemA es igual a indice.

5.5.12 MemoriaRAM.vhd.

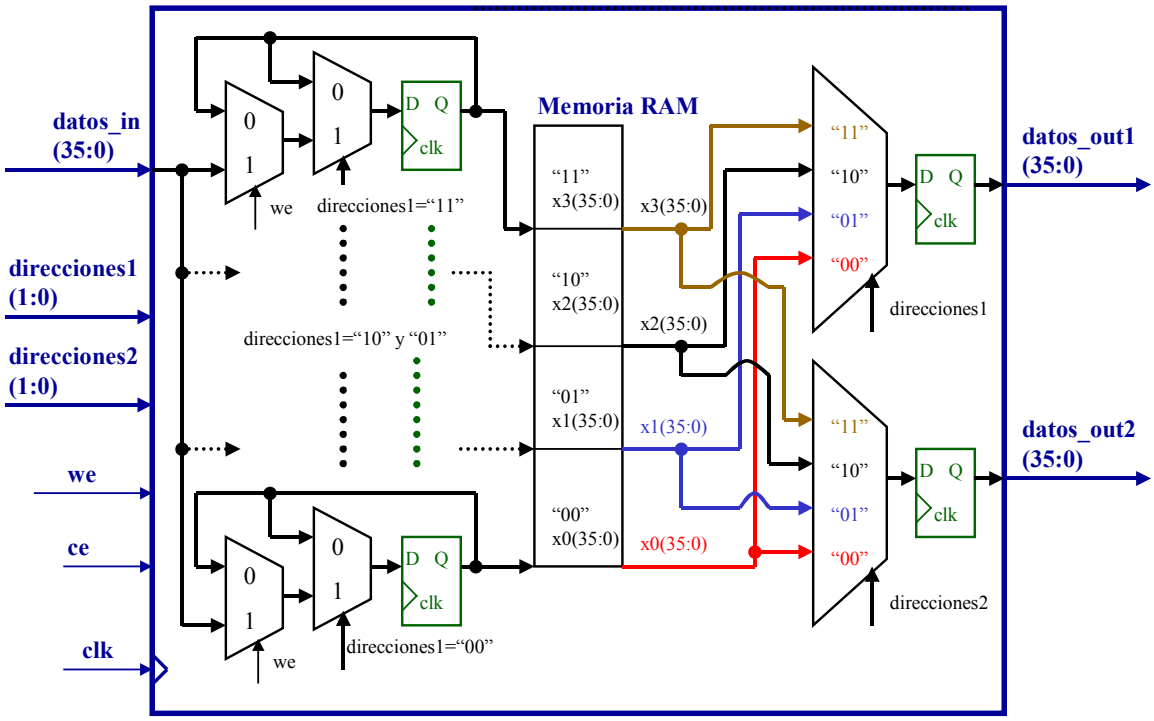


Figura 5.25: MemoriaRAM.vhd

- Memoria RAM de 4 filas con decoding depth bits cada una.
- La memoria tiene dos puertos de lectura y uno de escritura.
- Tanto la lectura como la escritura son síncronas, con latencia de 1 T_{CLK}.
- Si we = '1', entonces se escribe en la fila de la RAM indicada por direcciones1, el valor de datos_in, con latencia 1 T_{CLK}.
- datos_out1 <= lectura del contenido de la fila indicada por direcciones1, con latencia 1 T_{CLK}.
- datos_out2 <= lectura del contenido de la fila indicada por direcciones2, con latencia 1 T_{CLK}.
- El módulo se instancia 32 veces en el diseño. 16 veces para formar MemA y otras 16 para MemB. MemA y MemB contienen 64 filas, una para cada estado.

Esta memoria es independiente de la tecnología y multiplataforma. Se puede implementar en cualquier FPGA o dispositivo lógico programable con VHDL, porque contiene únicamente sentencias genéricas en VHDL. No instancia ninguna primitiva ni referencia a arquitecturas o recursos específicos de una FPGA o tarjeta de desarrollo.

Nota 5.5: La lectura se realiza siempre, independientemente del valor de we. En el caso de que we valga '1', entonces datos_out1 <= datos_in, porque la lectura de la memoria se realiza después de la escritura.

Para desarrollar este módulo utilizamos los ejemplos que aparecen en la guía " *Xilinx XST User Guide*", referencia [12]. En esa guía indican las opciones para diseñar una RAM con VHDL independiente de la tecnología. No sólo para FPGAS de Xilinx, sino que son válidas para todas las FPGAs y dispositivos lógicos programables con VHDL, de cualquier fabricante.

No incluimos las señales $\overline{\text{aclr}}$ ni EnableIn. EnableIn no es necesaria en el módulo. Sin embargo si que es necesario que antes de iniciar la decodificación de una trama de entrada, la memoria interna del decodificador esté con todos sus bits a '0'. Pero para conseguir esto no hace falta una señal de $\overline{\text{aclr}}$. Porque al iniciar el sistema, tras el encendido de la alimentación, la memoria se inicializa siempre a '0'. Por tanto cuando llegue la primera trama de datos al decodificador, la memoria ya estará correctamente actualizada.

Y cuando termina una trama, la memoria se queda con todos sus bits a '0'. Por lo que queda inicializada para la siguiente trama. Esto ocurre porque CalculatePaths maneja la memoria RAM con una estructura FIFO. De tal manera que cuando se activa VaciarTrellis, CalculatePaths va enviando a ExitDatoOut los bits almacenados en la columna cero de cada una de las filas. Desplaza los bits en cada fila una posición, y en la casilla que queda vacía, la (decoding depth-1), sitúa un '0'.

De los ejemplos de [12], la opción que mejor se adapta a nuestro diseño es la de dos puertos de lectura y uno de escritura. Lectura y escritura síncronas. La escritura debe ser siempre síncrona, pero la lectura podría ser asíncrona. Sin embargo hemos descartado esta opción porque el diseño resultante no es óptimo, ocupa un área mucho mayor.

Descartamos la lectura asíncrona porque al sintetizar aparece esta información:

"*INFO:Xst:1442 - HDL ADVISOR - The RAM contents appears to be read asynchronously. A synchronous read would allow you to take advantage of available block RAM resources, for optimized device usage and improved timings. Please refer to your documentation for coding guidelines.*"

Además hemos probado las dos opciones, y como podemos ver a continuación, la opción que hemos elegido de lectura síncrona es la correcta.

Tabla 5.15: Síntesis CalculatePaths.vhd. Comparativa RAM lectura síncrona/asíncrona. FPGA VIRTEX 4 XC4VSX55 10FF1148.

	Lectura síncrona.			Lectura asíncrona	
Logic Utilization	Used	Available	Utilization	Used	Utilization
Number of Slices	2454	24576	9%	4756	19 %
Number of Slice Flip Flops	890	49152	1%	1875	3 %
Number of 4 input LUTs	4765	49152	9 %	7057	14 %
Minimum period/ Maximum Frequency	5,89ns	169,6MHz		5,807ns	172,19MHz

5.5.13 ExitDatoOut.vhd.

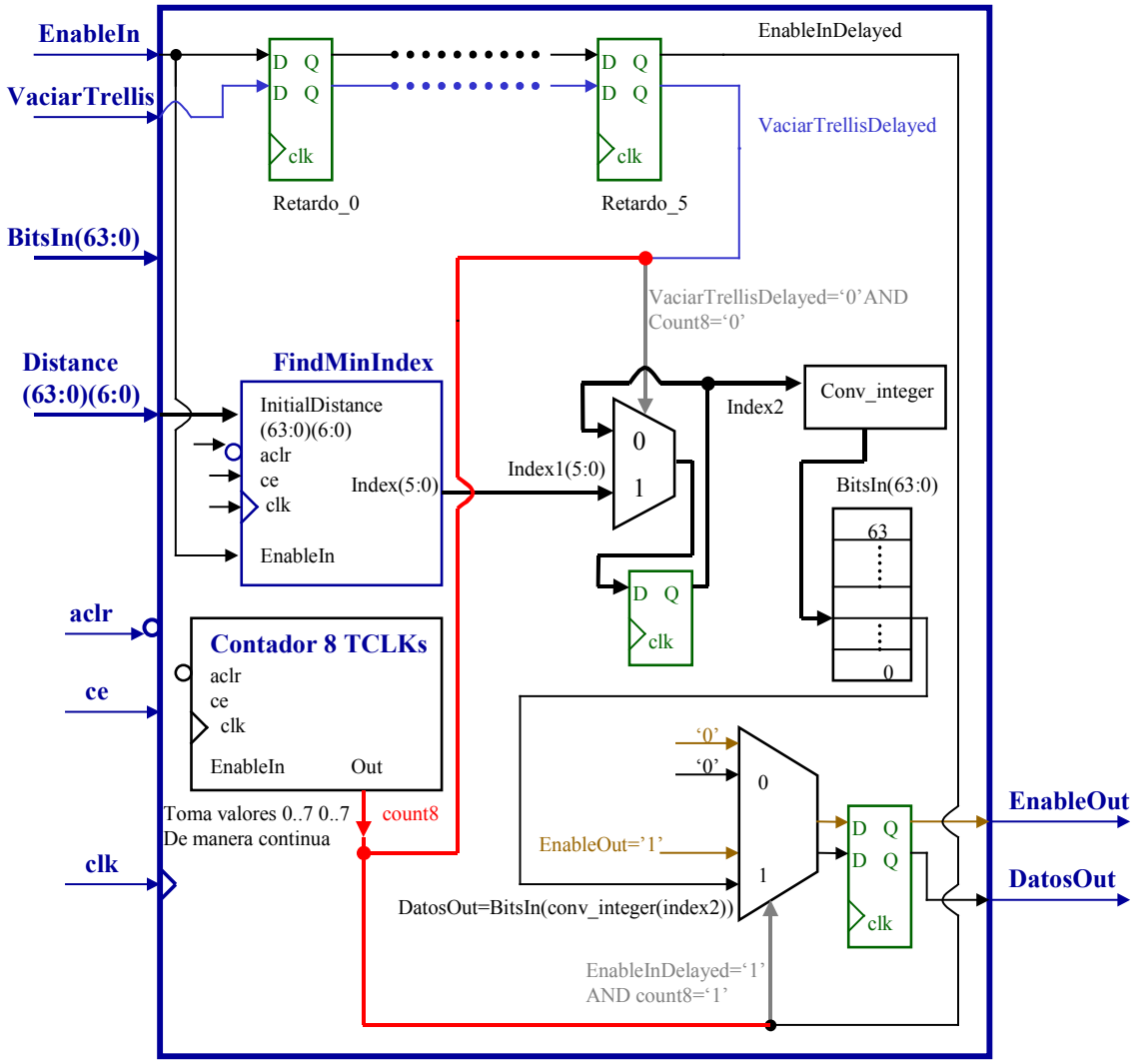


Figura 5.26: ExitDatoOut.vhd.

En la entrada del decodificador, llega un dato compuesto por dos bits en cada período de proceso. Tras una latencia (2+ (decoding depth +2)*8) T_{CLKs}, el módulo ExitDatoOut obtendrá en DatosOut el bit correspondiente a la decodificación de cada dato de entrada, y activará EnableOut durante 1 T_{CLK}.

EnableOut sólo se mantiene activo durante 1 T_{CLK} en cada período de proceso, y a '0' durante los 7 T_{CLKs} restantes. Lo hemos diseñado así para que al simulador le resulte más sencillo contar el número de bits decodificados en la salida.

La latencia entre EnableIn y EnableOut en este módulo es de 8 T_{CLKs}. El período de proceso de un dato es de 8 T_{CLKs}.

El trabajo de ExitDatoOut consiste en elegir el bit ganador tras todo el proceso de decodificación. Ese bit corresponde al primer bit del camino superviviente ganador. *Ver apartado 5.5.9 y 5.5.10.*

ExitDatoOut analiza los 64 caminos supervivientes, encuentra el que tenga la distancia mínima en su última posición en t_x, y elige su primer bit como el ganador tras todo el proceso de decodificación. Para ello necesita estas dos entradas:

- Distance: Tabla con las distancias hasta llegar a cada uno de los 64 estados en la última posición t_x de la malla trellis.

- BitsIn: Contiene el primer bit, el que corresponde a la posición $t_{x-(DecodingDepth)}$, de cada uno de los 64 caminos supervivientes.

Para realizar el trabajo ExitDatOut encuentra el índice, index1, de la casilla del vector Distance donde se encuentra la distancia mínima. Para ello instancia al módulo FindMinIndex.

$$\text{Index1} = \min[i \in [0..63]]_{Distance(ix) = \min_{0 \le i \le 63}[Distance(i)] = MinDistance}$$

Index1 es el índice de la casilla ganadora en BitsIn, así que DatosOut <=BitsIn(index1).

Hay tres situaciones de trabajo diferentes dependiendo de EnableIn y de VaciarTrellis

1. EnableIn='0' ; VaciarTrellis='0'. Situación inicial, la memoria que contiene los caminos supervivientes aún no están completa. Por tanto la entrada BitsIn sólo contiene ceros y el módulo ExitDatoOut no trabaja.

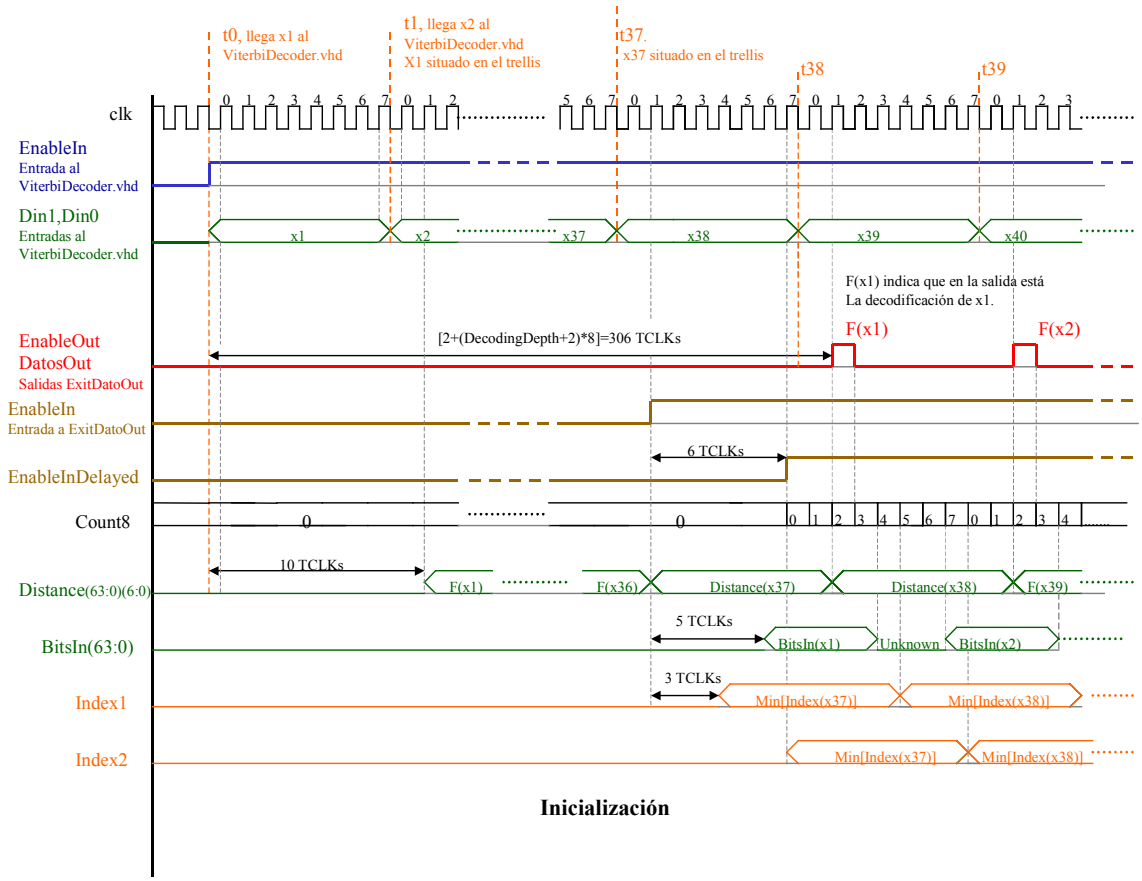
    - DatosOut = '0'.

2. EnableIn='1' ; VaciarTrellis='0'. Situación general, CalculatePaths envía en BitsIn el primer bit de cada uno de los caminos y ExitDatoOut saca en DatosOut el bit decodificado final.

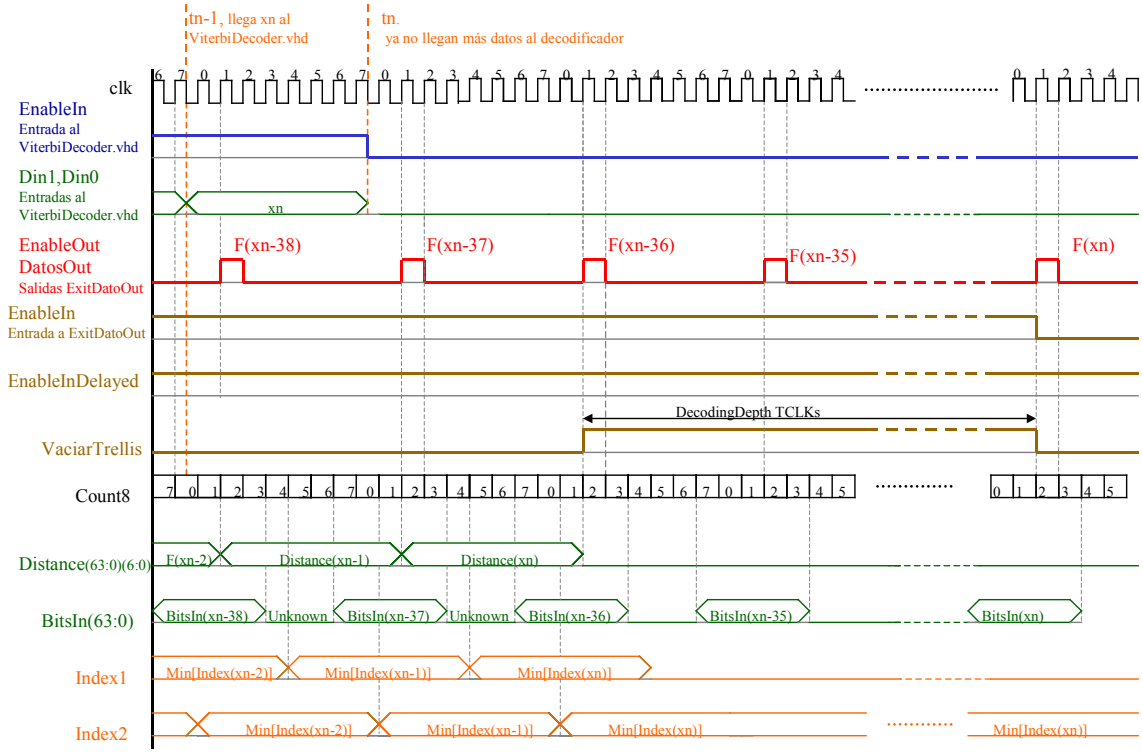
    - DatosOut = BitsIn(Index1)

3. EnableIn='1' ; VaciarTrellis='0'. En el momento t_n, finaliza la trama de entrada al decodificador, el último dato x_n ya estará situado en el trellis. Pero la decodificación no terminará hasta que se hayan decodificado los datos que aún quedan en la memoria del decodificador.
   En este caso lo que hay que hacer es seleccionar el último camino ganador, el que tenga una menor distancia en t_n. E ir sacando uno a uno sus decoding depth bits. Se continúa sacando un bit en cada período de proceso, por lo que permaneceremos en esta situación durante decoding depth períodos de proceso.
   Para realizar el trabajo, ExitDatoOut almacena en Index2 el índice del último camino ganador en t_n y en cada uno de los decoding depth períodos de proceso asigna:

    - DatosOut <=BitsIn(Index2).



Inicialización



Caso general y fin de trama

Cronograma 5.10: ExitDatoOut.vhd.
Decoding depth = 36.

5.6 Referencias.

[1] "Example: Viterbi algorithm". Application report: "Viterbi Decoding Techniques in the TMS320C54x Family". Noviembre 2010.
http://es.scribd.com/doc/43577105/Viterbi-Example

[2]T.K. Truong; Ming-Tang Shih; Irving S. Reed and E.H Satorius. "A VLSI Design for a Trace-Back Viterbi Decoder". IEEE Transactions on Communications, Vol 40, Nº3, pp 616-624, March 1992.
http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=135732

[3]W. Cary Huffman and Vera Pless. "Fundamentals of error Correcting Codes". Cambridge University Press, pp 546-557, 1 Febrero 2003.

[4]Alain Glavieux. "Channel Coding in Communication Networks. From Theory to Turbocodes". ISTE Ltd, pp129-137. 2007.

[5]Wei Chen. "RTL implementation of Viterbi decoder". Master Tesis desarrollada en Sistemas Electrónicos, departamento de Ingeniería de Computación en la Universidad de LinKöpings. Páginas 21-22 . 2 Junio 2006.
http://liu.diva-portal.org/smash/get/diva2:22064/FULLTEXT01

[6]Robert H. Morelos-Zaragoza. "The Art of error Correcting Coding". Wiley, pp 108-109 , segunda edición 2006.

[7]Syed Shahzad Shah; Saqib Yaqub and Faisal Suleman. " Part one: Viterbi codecs". Self-correcting codes conquer noise .15 Febrero 2001.
ftp://ftp.radionetworkprocessor.com/pub/reed-solomon/viterbi-chameleon.pdf

[8]Hui-Ling Lou. "Implementing the Viterbi Algorithm". Fundamentals and real-time issues for processor designers. IEEE Signal Proccesing Magazine. Page 47. September 1995.
http://www.mendeley.com/research/implementing-the-viterbi-algorithm-1/

[9]Frank Vahid. "Digital Design with RTL Design, Verilog and VHDL". John Wiley and Sons Publishers, pp 377-380, 2011.

[10] "Pipelining". ECEN 6263 Advanced VLSI Design, 7 Noviembre 2006.
http://lgjohn.okstate.edu/5263/lectures/pipe.pdf

[11] Suryanarayana B. Tatapudi and José G. Delgado-Frias. "Designing Pipelined Systems with a Clock Period Approaching Pipeline Register Delay". School of Electrical Engineering and Computer Science. Washington State University
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.3074&rep=rep1&type=pdf.

{statapud, jdelgado}@eecs.wsu.edu

[12] Documento Xilinx: "XST User Guide". Ug627, v11.3. September 16, 2009. Pages 126-188.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf

CAPÍTULO 6

6. IMPLEMENTACIÓN SIMULADORES.

6.1 Objetivos.

El desarrollo de un buen simulador es una parte fundamental en el diseño de módulos hardware. También es importante definir los vectores de test de entrada al módulo, de manera que se consiga abarcar todas las combinaciones posibles en los datos de entrada. La documentación sobre los tipos de simulación, cómo realizarlas y las herramientas necesarias está en: [1], [2] y [3]. Básicamente debemos realizar dos tipos de pruebas:

1. Utilizamos vectores de test que cubran todas las posibles combinaciones definidas como correctas en la entrada. De esta manera nos aseguramos de que no hay errores en el funcionamiento habitual.

2. Probamos con condiciones en la entrada que teóricamente no deberían darse nunca. De esta manera comprobamos cuál es el comportamiento del módulo ante casos excepcionales e inesperados. Así evitamos errores debidos a cualquier suceso no previsto en la entrada. Por ejemplo, hemos definido que antes de una trama hay que activar $\overline{aclr}$ durante al menos 8 T_{CLKs}. Mediante este tipo de pruebas vemos que ocurre si no se cumple esta condición.

La utilidad del simulador es que nos permite probar el funcionamiento de los bloques utilizando únicamente el PC, lo que supone un ahorro de tiempo y de recursos. Porque no es necesario pasar físicamente el diseño a la placa para realizar las comprobaciones.

En nuestro proceso de diseño desarrollamos un simulador que nos permite realizar simulación funcional y post place & route. Gracias a él podemos verificar que el decodificador Viterbi es funcionalmente correcto.

La simulación en el PC es muy exacta y nos asegura prácticamente al 100 % que al pasar el diseño a la placa este se comportará igual que en la simulación. Sin embargo no lo podemos garantizar de manera absoluta. Para estar totalmente seguros la única opción es probarlo físicamente en la placa mediante emulación.

El proceso de prueba real en el laboratorio es lento y pesado. Mientras que la simulación en el PC es muy rápida y cómoda, puesto que no se necesita más que el propio PC. Nosotros hemos utilizado esta ventaja para realizar unas pruebas muy exhaustivas, incluyendo todas las combinaciones posibles en la entrada, todos los casos posibles de error y todas las situaciones excepcionales e inesperadas.

En el conjunto de pruebas hemos necesitado más de 3000 horas de computación. Pero esto no supone un gran coste de tiempo en el desarrollo del proyecto. Porque una vez diseñado el simulador y los vectores de test, ya solamente hay que lanzar el simulador, dejar que el PC trabaje las horas que sean necesarias, (sin supervisar la simulación), y luego estudiar los resultados. Esto en cambio sería inviable realizarlo en el laboratorio con la tarjeta real, el generador de funciones y el osciloscopio.

En el *capítulo 7* detallamos todos los resultados que hemos obtenido, pero podemos adelantar que han sido los esperados. No hemos encontrado ningún fallo en la arquitectura ni el funcionamiento del decodificador que implementamos en el proyecto, el *ViterbiDecoder.vhd*. Realiza exactamente la decodificación Viterbi, con el mismo comportamiento que los decodificadores Viterbi comerciales.

6.2 Simuladores realizados.

Hemos realizado múltiples diseños a lo largo del proyecto. En este apartado explicamos la organización de todos ellos. Nos centramos únicamente en los archivos principales, la lista completa con todos los ficheros con código fuente está en el *anexoA*.

Nuestra especificación es la de implementar un decodificador Viterbi con decoding depth = 36. Sin embargo le damos un valor añadido, que consiste en que se puede implementar cualquier decoding depth con sólo cambiar una constante. Por este motivo en el diseño del simulador hemos tenido en cuenta esta característica y la hemos mantenido. De manera que un mismo simulador sirve para un decodificador Viterbi con cualquier decoding depth, con sólo variar una constante.

Aprovechando este valor añadido, hemos realizado un total de 5 decodificadores con decoding depth de 24, 32, 36, 48 y 60. Todos ellos los hemos simulado exhaustivamente para así comparar las características entre ellos. Para ello hemos utilizado el mismo simulador, cambiando solamente una constante.

Una característica del proyecto consiste en que debemos implementar un decodificador Viterbi en VHDL, y luego pasarlo a System Generator para integrarlo en un sistema WiMAX. Volvemos a insistir en que nuestro decodificador no es exclusivo para el protocolo WiMAX, referencias en *apartado 1.9.3*, sino que vale para cualquier aplicación en la que sea necesario decodificar un código convolucional.

Entonces no es suficiente con el simulador en VHDL, también debemos desarrollar otro en System Generator. Así nos aseguramos de que al integrar todos los bloques, el sistema sigue funcionando. Para optimizar el trabajo hicimos en primer lugar el de VHDL, dándole el valor añadido de que sus bloques son reutilizables en el simulador de System Generator. De manera que al realizar el simulador de System Generator no partimos de cero, sino que ya tuvimos una parte muy importante del trabajo hecho.

6.2.1 Simuladores codificados con VHDL.

TestSimulacionCompleta.vhd, en el que los bits de entrada al simulador se obtienen mediante un generador pseudoaleatorio.

TestLeeFichero.vhd en el que los bits de entrada se leen del fichero *BitsSimulacion.txt*.

Los 2 valen para cualquier decoding depth y para los dos decodificadores que hemos desarrollado: el UC3M ,(*ViterbiDecoder.vhd*), y Opencores (*decoderverilog.v*).

- *TestLeeFichero.vhd* se adapta automáticamente a cualquier decoding depth. No hay que modificar nada.

- En *TestSimulacionCompleta.vhd* sólo hay que modificar la constante EsperaTramasMin, que debe ser >= decoding depth + 4. Interesa que EsperaTramasMin tenga el valor mínimo posible, porque así la simulación es más rápida. Pero también se puede poner un valor más alto, ejemplo 64, de manera que con el mismo simulador, sin modificar nada, se puede simular cualquier *ViterbiDecoder.vhd* que tenga un decoding depth <= 60.

El módulo raíz de la simulación es *SimulacionCompleta.vhd*, decodificador UC3M, o *SimulacionCompletaVerilog.v* para el Opencores. Este bloque raíz incluye el decodificador, *ViterbiDecoder.vhd* o *decoderverilog.v*, y otros bloques auxiliares necesarios para realizar la simulación. Podríamos haber incluido los bloques auxiliares en el propio simulador, de manera que el archivo raíz fuese *ViterbiDecoder.vhd* o *decoderverilog.v*. Sin embargo hemos descartado esta opción porque al implementar los bloques auxiliares en módulos independientes, podemos pasarlos inmediatamente a System Generator. En cambio si estuviesen incluidos en el propio simulador no podríamos reutilizarlos de ninguna manera al pasar a System Generator.

Otra ventaja más de los 2 simuladores es que nos permiten simular el decodificador UC3M y el de Opencores sin hacer cambios en su estructura. Únicamente hay que hacer un cambio inevitable, en el caso del UC3M hay que instanciar el módulo raíz *SimulaciónComplet.vhd*, y en el de Opencores el *SimulaciónCompletaVerilog.v*.

6.2.2. Simuladores codificados con System Generator.

Incluimos una mejora importante, que consiste en que con un mismo simulador simulamos 2 decodificadores al mismo tiempo. El UC3M, *ViterbiDecoder.vhd,* y el que tomamos como referencia, IP core Viterbi decoder v5.0 de Xilinx, datasheet en [16A]. A los dos les llegan exactamente los mismos símbolos de entrada. De manera que podemos compararlos al mismo tiempo con total exactitud, puesto que las condiciones de la prueba son las mismas para ambos.

Esta mejora resulta fundamental porque los dos decodificadores implementan el mismo algoritmo, así que los dos deben tener el mismo comportamiento. Utilizando este simulador nos hemos asegurado de que el *ViterbiDecoder.vhd* se comporta igual que el IP core de Xilinx. Entonces podemos verificar que nuestro decodificador funciona correctamente, cumpliendo fielmente el algoritmo de Viterbi.

ViterbiDecoderUC3M_Xilinx_Depth36_Simulacion_K7R05_171_133.mdl.
Es el simulador definitivo, simula al mismo tiempo el *ViterbiDecoder.vhd* y el IP core Viterbi de Xilinx.

Es suficiente con utilizar el modelo definitivo porque tiene todo lo que se necesita. Sin embargo también incluimos el código de los simuladores intermedios que hemos utilizado antes de llegar al definitivo. Todos están terminados y funcionan correctamente, así que pueden utilizarse aunque incluyen menos funcionalidades que el definitivo. Porque los modelos intermedios sólo simulan el decodificador UC3M, no incluyen el de Xilinx. Pero tienen una ventaja, y es que como sólo instancian un decodificador, la simulación es más rápida. Esto es de gran utilidad cuando es necesario realizar una simulación muy exhaustiva con millones de símbolos de entrada.

a) **ViterbiDecoderUC3M_Depth36_Simulacion_K7R05_171_133.mdl**
Simulación y síntesis de *ViterbiDecoder.vhd*. Este simulador trabaja con una sola frecuencia de reloj: $F=1/T_{CLK}$, de manera que el decodificador UC3M emplea $8T_{CLKs}$ en decodificar un dato.

b) **ViterbiDecoderUC3M_Depth36_8TCLK_TCLK_K7R05_171_133.mdl**
Simulación y síntesis de *ViterbiDecoder.vhd*. Este simulador trabaja con dos frecuencias de reloj: $F_2=1/T_2=1/(8T_{CLK})$ y $F_1=1/T_1=1/T_{CLK}$.
El decodificador UC3M trabaja con la frecuencia $F_1=8F_2$. De esta manera en un sólo período T_2 es capaz de decodificar un dato.
El decodificador de Xilinx trabaja con F_2.

c) **ViterbiDecoderUC3M_Depth36_SoloSintesis_K7R05_171_133.mdl**
Síntesis de *ViterbiDecoder.vhd*. Sólo incluye el bloque del decodificador. De esta manera se puede ver el área que ocupa el bloque en solitario. Este archivo no es un simulador.

La diferencia entre los simuladores "a" y "b" está en la frecuencia y formato con la que le llegan los símbolos al *ViterbiDecoder.vhd*. En el modelo "b" esa frecuencia y formato coincide exactamente con la que precisa el decodificador IP core de Xilinx en su entrada. En cambio en el "a" las entradas al *ViterbiDecoder.vhd* no serían válidas para el IP core de Xilinx.

En el simulador definitivo los símbolos de entrada al decodificador UC3M y al de Xilinx deben ser exactamente iguales. Por eso para llegar al modelo definitivo partimos del simulador "b", que sólo incluye el *ViterbiDecoder.vhd* y le instanciamos también el IP core de Xilinx.

El simulador "a" es el que implementamos en primer lugar, importando a System Generator los bloques del simulador realizado en VHDL. Después vimos que era necesario añadirle mejoras y por eso partimos del modelo "a" para llegar al "b".

Un valor añadido de nuestros modelos es que aceptan cualquier decoding depth en los decodificadores que instancian. Al cambiar el decoding depth, la estructura de los simuladores es la misma, sólo hay que cambiar la constante ACLR_Desactivado. *Ver apartado 6.8.2.* Interesa adaptar la constante del simulador para optimizar el tiempo de simulación. Pero también existe la posibilidad de fijar un valor alto en esa constante, de manera que sea válida para cualquier decoding depth, a cambio de que el tiempo de la simulación aumenta algo porque no está optimizado.

Hemos realizado un total de 5 decodificadores con decoding depth de 24, 32, 36, 48 y 60. Para cada uno de ellos están los 4 modelos completamente operativos: el definitivo, el "a", el "b" y el "c". Entonces en total tenemos 20 modelos completos. Se identifica fácilmente el decoding depth del decodificador instanciado por el propio nombre del simulador.

Es mucho más cómodo tener almacenado un simulador específico para cada decoding depth, así no hay que preocuparse de que decodificador se está instanciando en cada

momento. Las diferencias entre ellos sólo son en algunas constantes y en el decodificador que instancian.

Para optimizar el trabajo, otro valor añadido es que la misma estructura nos vale para simular el decodificador de Opencores. El único cambio que hay que hacer es que en este modelo se instancia el *decoderverilog.v* en vez del *ViterbiDecoder.vhd*. El nombre que le hemos dado es:

ViterbiDecoderUC3M_Opencores_Depth32_Simulacion_K7R05_171_133.mdl.
Este es el modelo definitivo, que simula al mismo tiempo el *decoderverilog.v* y el *ViterbiDecoder.vhd.*

6.3 Arquitectura básica.

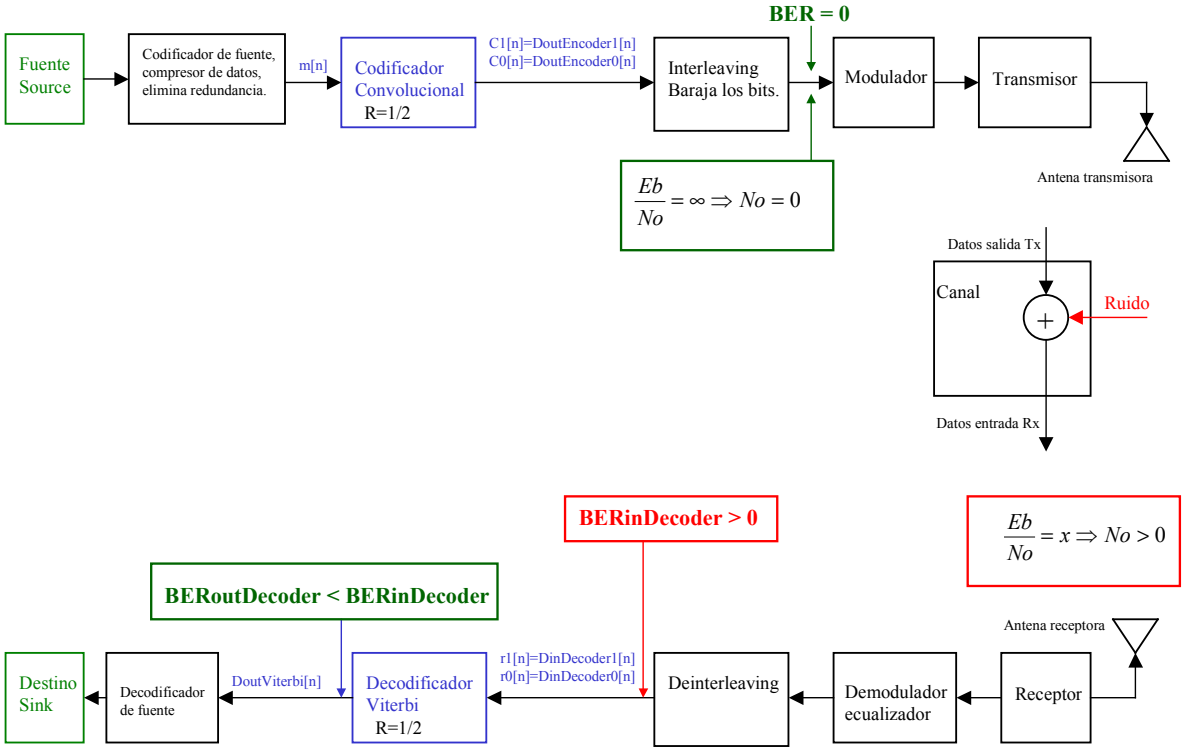


Figura 6.1: Descripción sistema de comunicaciones.
Referencias en [4], [5A], [6A], [7A], [8], [9], [10] y [11A].

1. La secuencia con la información fuente se denomina m[n]. Esa secuencia es la entrada al codificador.

2. La secuencia se codifica obteniendo c[n] = (DoutEncoder$_1$[n], DoutEncoder$_0$[n]).

3. c[n] pasa por el *interleaving* (entrelazador), modulador y transmisor después se transmite.

4. El canal de transmisión añade ruido, de manera que al final del proceso de recepción se obtiene la secuencia r[n]=(DinDecoder$_1$[n],DinDecoder$_0$[n]) que es diferente de c[n].

5. $$BERinDecoder = \frac{\text{Num bits diferentes}(c_1[n], r_1[n]) + \text{Num bits diferentes}(c_0[n], r_0[n])}{2 * \text{Número de bits}(c_1[n])}$$

6. $$SERinDecoder = \frac{Símbolosdiferentes(c[n], r[n])}{NumSimbolos(c[n])} = \frac{Símbolosdiferentes(c[n], r[n])}{NumBits(c_1[n])}$$

7. El decodificador decodifica la secuencia r[n], obteniendo DoutViterbi[n]. En un sistema ideal sin ruido, tendremos r[n] = c[n] → BERinDecoder = 0. Con estas condiciones debe cumplirse obligatoriamente que DoutViterbi[n] sea exactamente igual a m[n-retardo], y por tanto BERoutDecoder=0. Retardo es la latencia del sistema entre la entrada al codificador y la salida del decodificador.

8. Pero en un sistema real con ruido, los bits se habrán modificado, de manera que tendremos c[n] ≠ r[n] y BERinDecoder>0. Con estas condiciones el decodificador debe disminuir los errores que el ruido ha añadido a la secuencia original. Entonces se debe obtener BERoutDecoder < BERinDecoder. La secuencia final DoutViterbi[n] debe ser lo más parecida posible a la inicial: m[n-retardo].

9. $$BEROutDecoder = \frac{Bitsdiferentes(m[n - retardo], DoutViterbi[n])}{NumBits(m[n])}$$

Para asegurarnos de que el *ViterbiDecoder.vhd* y el *decoderverilog.v* funcionan correctamente diseñamos un simulador que modela el sistema de comunicaciones de la figura anterior.

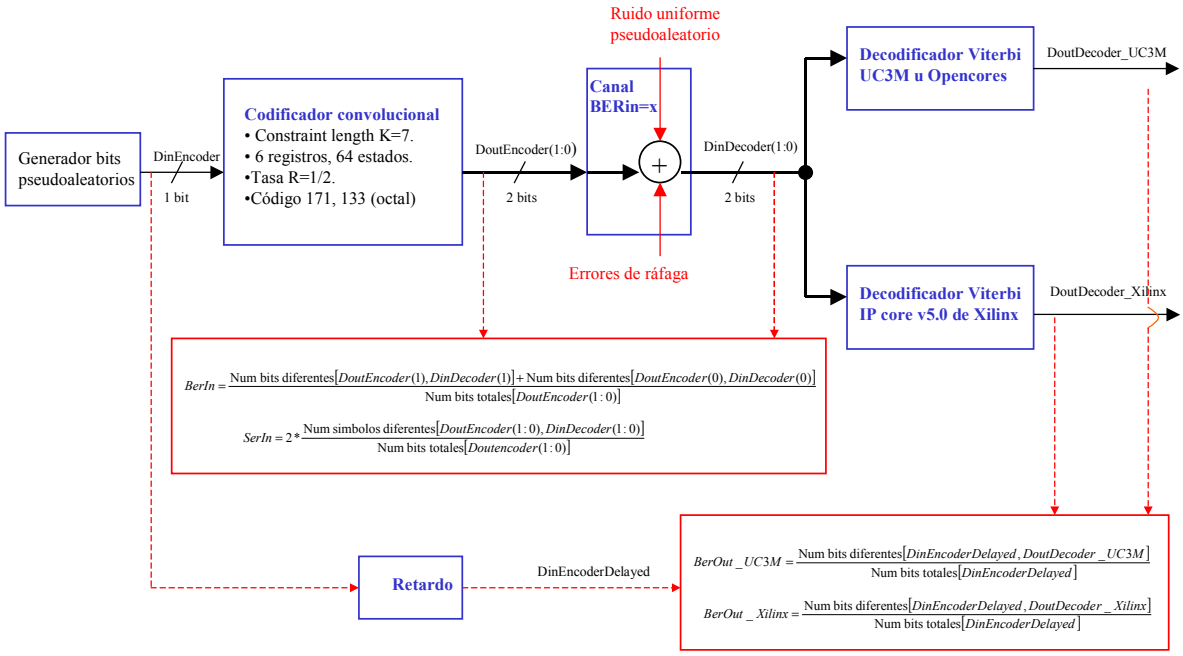


Figura 6.2: Arquitectura básica simulador.

En el simulador no es necesario incluir todos los bloques del sistema de comunicaciones de la *figura 6.1*. Los bloques entrelazador, modulador y transmisor no aportan nada útil al simulador. Porque en caso de ser ideales su función se anula con la de sus bloques inversos: desentrelazador, demodulador y receptor. De manera que no modifican nada la secuencia de bits transmitida. En el caso real el transmisor y el receptor son ruidosos, pero ese efecto del ruido lo podemos modelar con la fuente de ruido del canal, no aportaría ninguna ventaja utilizar dos fuentes de ruido diferentes en el simulador. La técnica común es la que haremos nosotros y consiste en modelar todo el ruido del sistema con una única fuente de ruido, la del canal.

Hemos desarrollado simuladores en VHDL para verificar el comportamiento de *Viterbidecoder.vhd* y *decoderverilog.v*. Y también otros en System Generator para simular los 2 bloques anteriores cuando los importemos a System Generator. En todos los casos el funcionamiento básico es el mismo y consiste en:

1. Se generan unos bits pseudoaleatorios (cambiando lógicamente la semilla en cada simulación). No utilizamos un modelo aleatorio sin semillas de inicialización, porque no está disponible en VHDL ni en System Generator. Utilizamos el bloque Uniform Random Number, su datasheet está en la ayuda de Matlab-Simulink.

2. Esos bits se codifican convolucionalmente, obteniéndose la secuencia DoutEncoder(1:0), que es la que se transmite por el canal.

3. El ruido del canal de transmisión modifica los bits, de manera que la secuencia recibida en el receptor, DinDecoder(1:0), es diferente a la transmitida DoutEncoder(1:0).

4. Para modelar el ruido utilizamos un generador uniforme pseudoaleatorio, con el que se obtiene una distribución uniforme de bits erróneos en DinDecoder(1:0). La cantidad de bits erróneos la fija una constante, BERin = x, por lo que es inmediato variar el nivel de ruido al realizar diferentes simulaciones.

5. Se pueden incluir errores de ráfaga en DinDecoder. La duración e intervalo entre ráfagas es variable e inmediata de caracterizar, porque depende únicamente de constantes. También es posible utilizar al mismo tiempo errores de ráfaga y errores uniformemente distribuidos con BERin = x.

6. La secuencia recibida es la entrada del decodificador Viterbi. Este decodifica la secuencia, obteniendo DoutDecoder, que debe ser lo más parecido posible a DinEncoderDelayed.

7. El retardo es igual a la suma de todos los retardos del sistema de comunicaciones: retardo= $\text{latencia}_{\text{codificador}} + \text{retardo}_{\text{canal}} + \text{latencia}_{\text{decodificador}}$.

8. El simulador calcula en tiempo real todos los parámetros de interés y los muestra en la consola.

6.4 Modelado del ruido.

La documentación de referencia de este apartado está en [5B], [6B], [7B] y [11B].

6.4.1 Ecuaciones matemáticas.

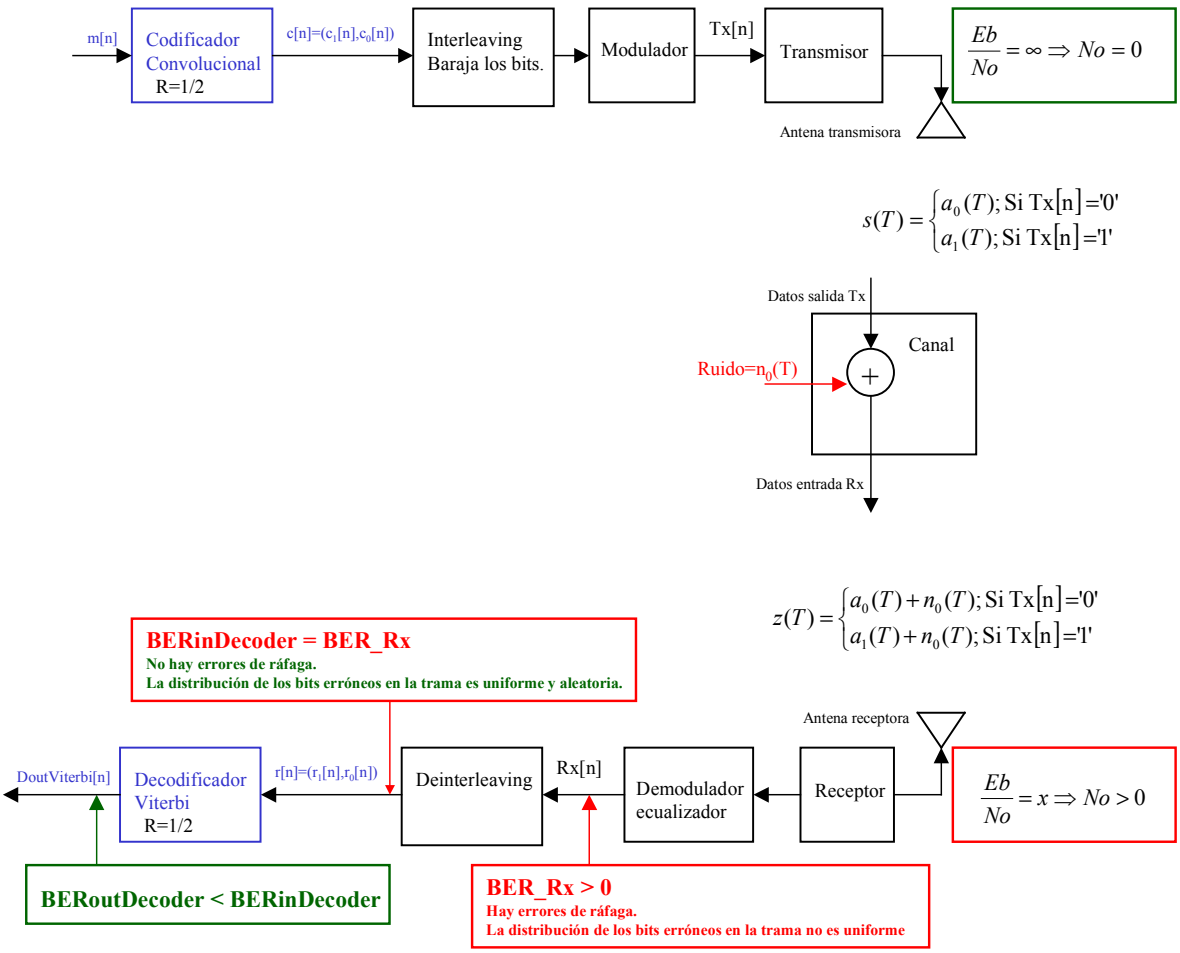


Figura 6.3: Efectos del ruido en el sistema.

En el módulo canal incluimos un generador de ruido uniforme pseudoaletorio que modela todo el ruido del sistema de comunicaciones.

Nos basamos en un sistema en el que el ruido sea blanco y gaussiano. Canal AWGN, additive white gaussian noise.

Además el canal es BSC, binary symmetric channel. Esto significa que por el canal únicamente se transmiten bits, y la probabilidad de transmitir un '1' es la misma que la de transmitir un '0'. P(Tx='1')=P(Tx='0') = 0.5.

Con estas condiciones en el receptor tendremos una energía $z(T)= a_i(T)+ n_o(T)$; i=0 ó 1.
a_i es la energía del bit transmitido. a_0 en caso de transmitir un '0' y a_1 en caso de '1'.
n_o es el ruido añadido, y consiste en una variable aleatoria gaussiana de media cero.

La función densidad de probabilidad (pdf) del ruido gaussiano aleatorio n_0 es:

$$p(n_0) = \mathrm{N}(0,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-n_0^2}{2\sigma^2}}$$

Densidad de probabilidad normal o gaussiana de media cero y varianza σ^2.

La función densidad de probabilidad de z(T) cuando se transmite un '0' la llamamos $p(z|s_0)$. Y $p(z|s_1)$ cuando se transmite un '1'.

$$p(z \mid s_0) = \mathrm{N}(a_0,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(z-a_0)^2}{2\sigma^2}}$$

$$p(z \mid s_1) = \mathrm{N}(a_1,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(z-a_1)^2}{2\sigma^2}}$$

Entonces en el receptor tendremos esta densidad de probabilidad:

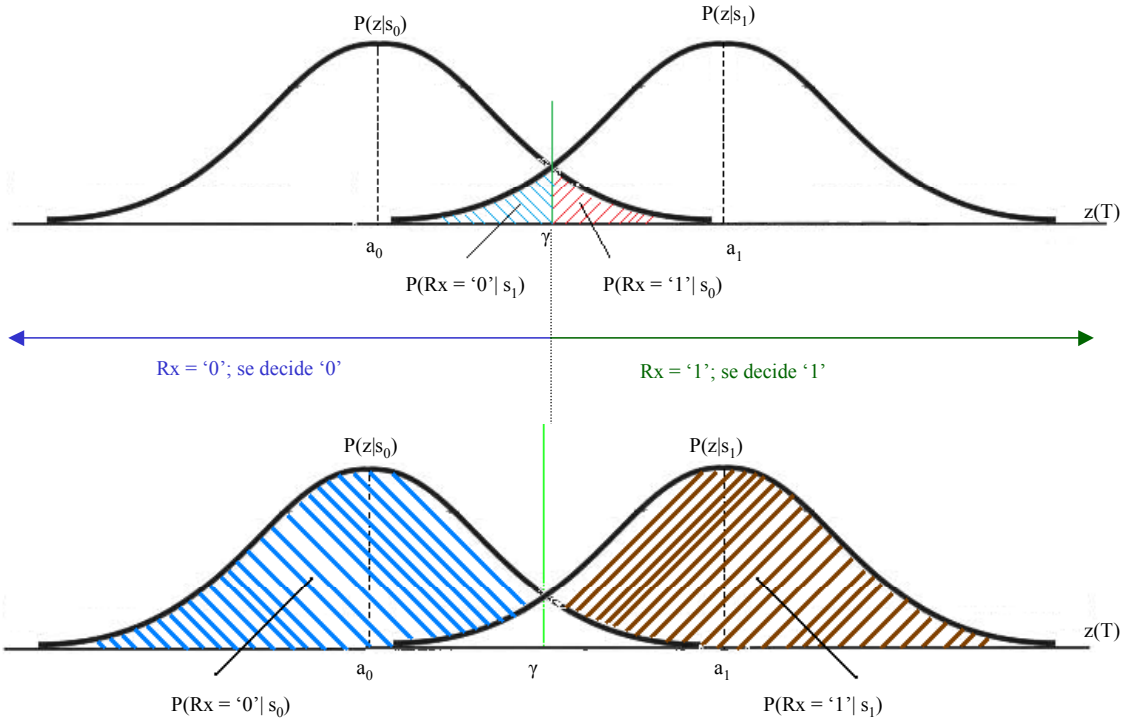


Figura 6.4: Función densidad de probabilidad en el receptor.

Con las condiciones canal AWGN, transmisión BSC y modulación BPSK ó QPSK, las ecuaciones que definen las probabilidades de decisión en el receptor son:

Probabilidad de que en el receptor se interprete '1' cuando se transmitió '0':

$$P(Rx = '1' \mid s_0) = \frac{1}{2} erfc\left(\sqrt{\frac{Eb}{No}}\right) = Q\left(\sqrt{\frac{2Eb}{No}}\right)$$

, la misma ecuación vale para BPSK y QPSK.

Eb es la energía del bit. En BPSK tenemos: $a_0 = -\sqrt{Eb}$, $a_1 = \sqrt{Eb}$

$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$; complementary error function or co-error function.

$\frac{1}{2} erfc(x)$; Es el área comprendida entre x e ∞ de una gaussiana de media cero y varianza 0.5.

$Q(x) = \frac{1}{2} erfc\left(\frac{x}{\sqrt{2}} \right)$; Área comprendida entre x e ∞ de una gaussiana de media cero y varianza 1.

El resto de probabilidades son inmediatas:

- Interpretar '0' cuando se transmitió '1': $P(Rx = '0' | s_1) = P(Rx = '1' | s_0)$
- Interpretar '1' cuando se transmitió '1' = interpretar '0' cuando se transmitió '0': $P(Rx = '1' | s_1) = P(Rx = '0' | s_0) = 1 - P(Rx = '1' | s_0)$

La probabilidad de bit erróneo P_b= BER:

$$BER = P(Tx = '1') * P(Rx = '0' | s_1) + P(Tx = '0') * P(Rx = '1' | s_0) =>$$
$$BER = \frac{1}{2} P(Rx = '0' | s_1) + \frac{1}{2} P(Rx = '1' | s_0) = \frac{1}{2} \left(P(Rx = '0' | s_1) + P(Rx = '1' | s_0) \right) =>$$

$$P_b = BER = P(Rx = '0' | s_1) = P(Rx = '1' | s_0)$$

Mediante la fórmula de la BER podemos diseñar el canal de manera sencilla, olvidándonos de las funciones densidad de probabilidad. El canal debe cumplir simplemente esto:

$$\text{Si } Tx[n_0] = '0' \Rightarrow \begin{cases} Rx[n_0] = '0' \text{ con probabilidad } 1 - P_b \\ Rx[n_0] = '1' \text{ con probabilidad } P_b \end{cases}$$

$$\text{Si } Tx[n_0] = '1' \Rightarrow \begin{cases} Rx[n_0] = '1' \text{ con probabilidad } 1 - P_b \\ Rx[n_0] = '0' \text{ con probabilidad } P_b \end{cases}$$

Aplicando la fórmula anterior sabemos cuántos bits erróneos habrá en cada trama. Estos errores se producen de manera aleatoria pero no uniforme. Es habitual que los errores se concentren en una parte de la trama formando errores de ráfaga. En [11C], [12], [13] y [14], está la documentación de referencia sobre errores de ráfaga, desvanecimientos en el canal y la técnica entrelazado/desentrelazado. Esta técnica se emplea para atenuar los efectos negativos producidos por los errores de ráfaga y desvanecimientos en el canal.

Ejemplo si Tx[n]= {1,0,1,0,1,0,0,1,0,1,1,1,0,1,0,1,0,1,0,1} y BER = 0,2, entonces en Rx tendremos 4 errores, que pueden agruparse en una ráfaga, por ejemplo:
Rx[n]= {1,0,1,0,1,0,0,0,1,0,0,1,0,1,0,1,0,1,0,1}

En la entrada del desentrelazador está Rx[n] con una BER determinada y una distribución no uniforme de los errores, puesto que se pueden organizar en ráfagas. Pero a la salida del desentrelazador, como se cambian de posición todos los bits de la trama, los errores de ráfaga se reparten de manera uniforme en la trama. Por eso la distribución de los bits erróneos es uniforme y aleatoria. Entonces tendremos por ejemplo:

r[n]= {1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,0,0,1,0}= {Tx[3], Tx[9], Tx[16], Tx[0], Tx[13], Tx[5], Tx[10] ,Tx[19], Tx[7], Tx[15], Tx[18], Tx[8],Tx[14], Tx[4], Tx[17], Tx[2],Tx[11], Tx[6], Tx[1],Tx[12]}

Por eso la BER a la entrada y salida del desentrelazador es la misma, pero no la distribución de los bits erróneos.

6.4.2 Implementación del ruido uniforme en el simulador.

Aquí definimos como se modela el ruido en r[n] = DinDecoder[n] de la *figura 6.3*.

Tenemos dos variables que definen el ruido, Eb/No y la BER. El objetivo del simulador es poder modelar todos los niveles de ruido. Para ello hay dos opciones posibles:

1. El simulador trabaja con la variable Eb/No de entrada al receptor. Esta variable podrá modificarse por el operador, de manera que se obtendrá BERoutDecoder en función de Eb/No.

2. Trabajar con BERinDecoder. El operador podrá modificar esta variable de manera que a partir de ella se obtenga BERoutDecoder.

BER y Eb/No se relacionan con una fórmula matemática que depende de la modulación y el tipo de ruido. Por ejemplo, hemos visto en el apartado anterior que para el caso de canal AWGN, transmisión BSC y modulación BPSK ó QPSK tenemos:

$$BERinDecoder = \frac{1}{2} erfc\left(\sqrt{\frac{Eb}{No}}\right) = Q\left(\sqrt{\frac{2Eb}{No}}\right)$$; *Ecuación 1.*

Elegimos la segunda opción porque tiene dos ventajas importantes:

1. La simplicidad, puesto que modelamos el ruido con un simple generador de ruido uniforme. En cambio trabajando con Eb/No hay que utilizar la fórmula erfc que es mucho más difícil de implementar en hardware.

2. Trabajando con Eb/No restringimos la utilidad del simulador a sólo un tipo de modulación, (BPSK ó QPSK), y ruido, (canal AWGN). En cambio con la opción que hemos elegido tenemos un importante valor añadido. Consiste en que el mismo simulador sin ningún cambio, sirve para otras modulaciones y tipos de ruido. Consultar [7C], para informarse sobre otras modulaciones y ruido.

Entonces el diseño del canal es extremadamente simple, se reduce a dos generadores uniformes pseudoaleatorios, dos comparadores y dos inversores.

Para cada bit de entrada al canal el generador obtiene un z_1, z_0 aleatorio y uniformemente distribuido entre 0 y 1. Pero el generador es pseudoaleatorio igual que el que el de la entrada del codificador. Por eso si ejecutamos 2 o más veces el simulador sin cambiar la semilla, ni BER=x, se obtendrá siempre el mismo ruido.

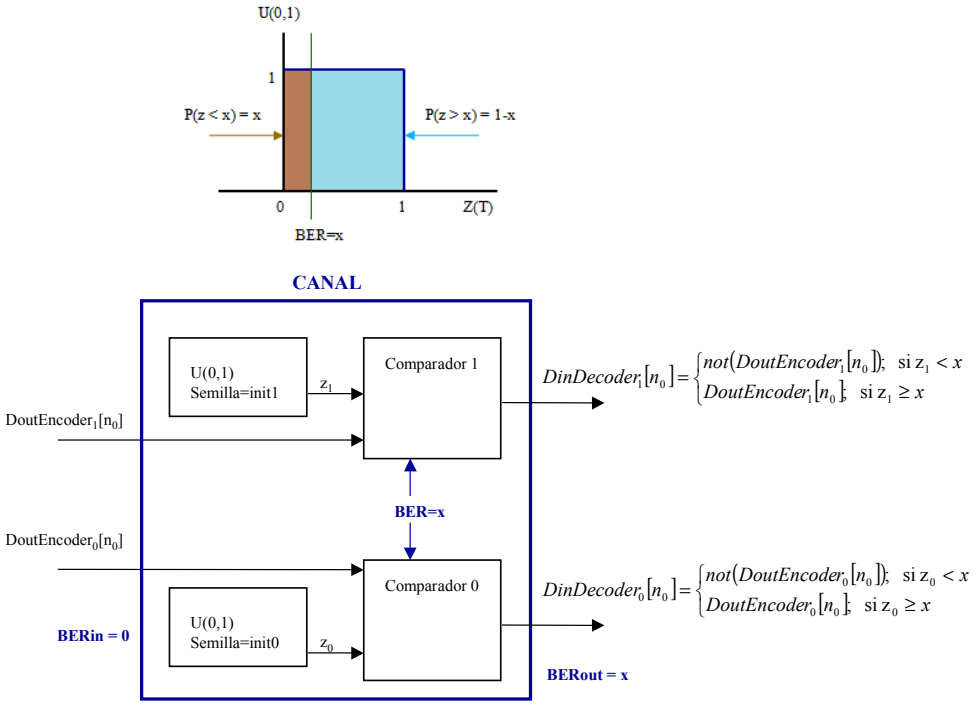


Figura 6.5: Implementación del canal ruidoso.

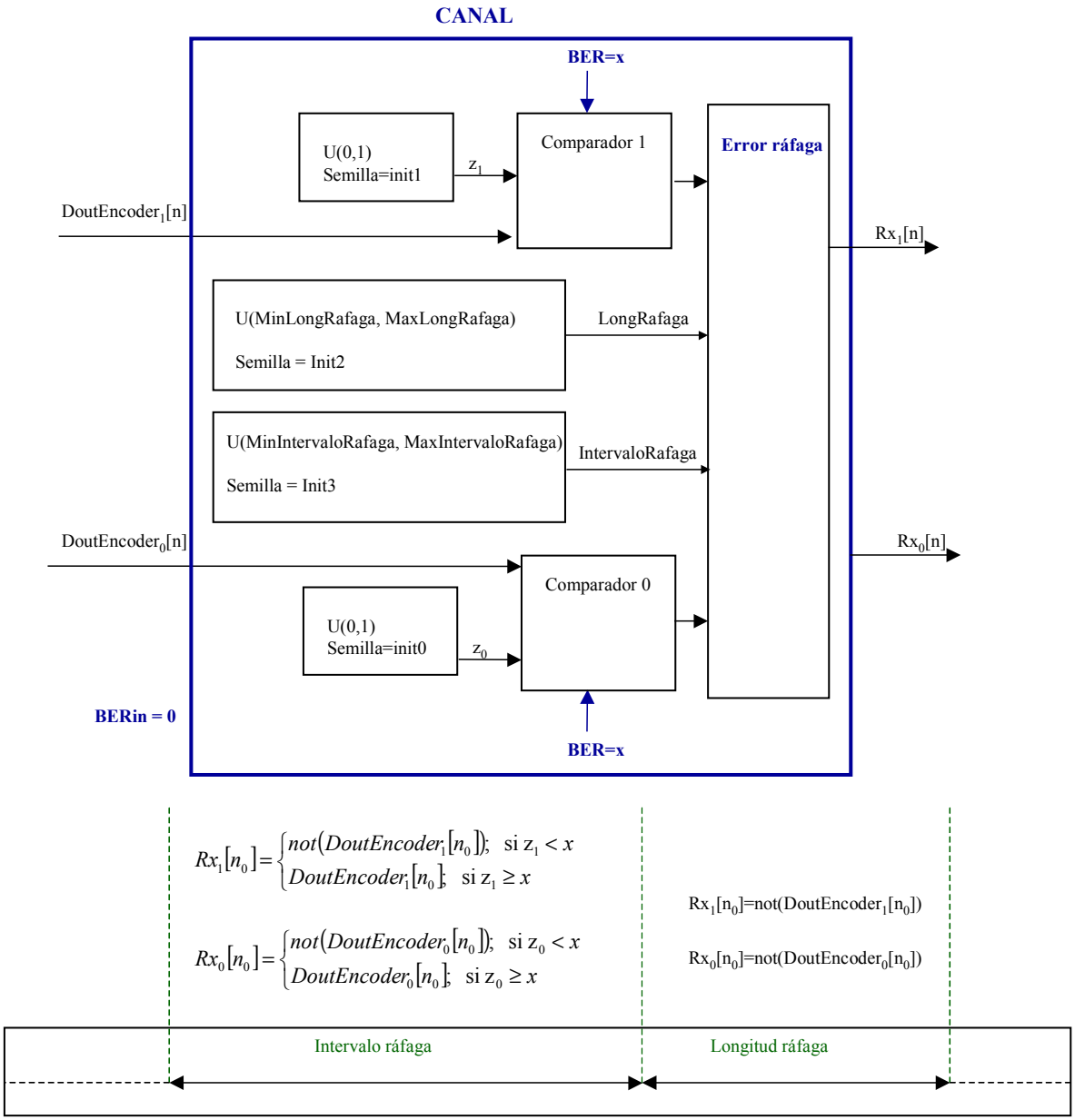
$$BERout = \frac{NumVeces(z_1 < x) + NumVeces(z_0 < x)}{2 * NumBits(DinDecoder_1[n])} \Rightarrow \begin{vmatrix} BERout = x \\ Con\ n\ suficientemente\ grande \end{vmatrix}$$

6.4.3 Implementación errores de ráfaga.

Aquí definimos como se modela el ruido en Rx[n] de la *figura 6.3*. Este ruido puede tener estas 3 componentes:

1. Errores en los bits distribuidos de manera uniforme y aleatoria en la trama.
2. Errores de ráfaga. Referencias [11C], [12], [13] y [14].
3. Una combinación de los dos anteriores.

En un sistema típico de comunicaciones los errores de ráfaga se eliminan con el desentrelazador, convirtiéndose en errores aleatorios uniformemente distribuidos. Pero nosotros los incluimos en el simulador para demostrar lo perjudicial que son los errores de ráfaga en el sistema, y así justificar la presencia obligada del entrelazador. Se trata por tanto de un valor añadido en el simulador. Lo incluimos únicamente en los de VHDL, por que sólo con ellos ya demostramos la necesidad de usar el entrelazador.



$$Rx_1[n_0] = \begin{cases} not(DoutEncoder_1[n_0]); & \text{si } z_1 < x \\ DoutEncoder_1[n_0]; & \text{si } z_1 \geq x \end{cases}$$

$$Rx_0[n_0] = \begin{cases} not(DoutEncoder_0[n_0]); & \text{si } z_0 < x \\ DoutEncoder_0[n_0]; & \text{si } z_0 \geq x \end{cases}$$

Figura 6.6: Implementación errores de ráfaga.

Mediante un generador uniforme pseudoaleatorio se determina la duración de las ráfagas. Cada ráfaga tendrá una duración aleatoria, uniformemente distribuida entre los límites MinLongRafaga y MaxLongRafaga. En el caso de que los dos límites sean iguales, la duración será siempre la misma e igual a MinLongRafaga=MaxLongRafaga.

Mediante otro generador uniforme pseudoaleatorio se determina el intervalo entre las ráfagas. Cada intervalo tendrá una duración aleatoria, uniformemente distribuida entre los límites MinIntervaloRafaga y MaxIntervaloRafaga. En el caso de que los dos límites sean iguales, el intervalo será siempre el mismo e igual a MinIntervaloRafaga.

$$BERout = \frac{2 * NumBitsRafaga + NumVeces\left|\begin{array}{l}[(z_1 < x) + (z_0 < x)] \\ \text{Dentro del intervalo entre ráfagas}\end{array}\right.}{2 * NumBits(Rx_1[n])} \Rightarrow$$

$$\left|\begin{array}{l} BERout = \dfrac{NumBitsRafaga + x * NumBitsIntervaloRafaga}{NumBits(Rx_1[n])} \\ \text{Con n suficientemente grande}\end{array}\right.$$

6.4.4 Gráficas BER en función de Eb/No.

En los decodificadores comerciales el rendimiento del decodificador se muestra utilizando gráficas como la de la *figura 6.7*. Documentación sobre estas gráficas en las referencias, (del *tema 7*), [5], [6], [7], [34], [35], [36], [37], [38] y [39].

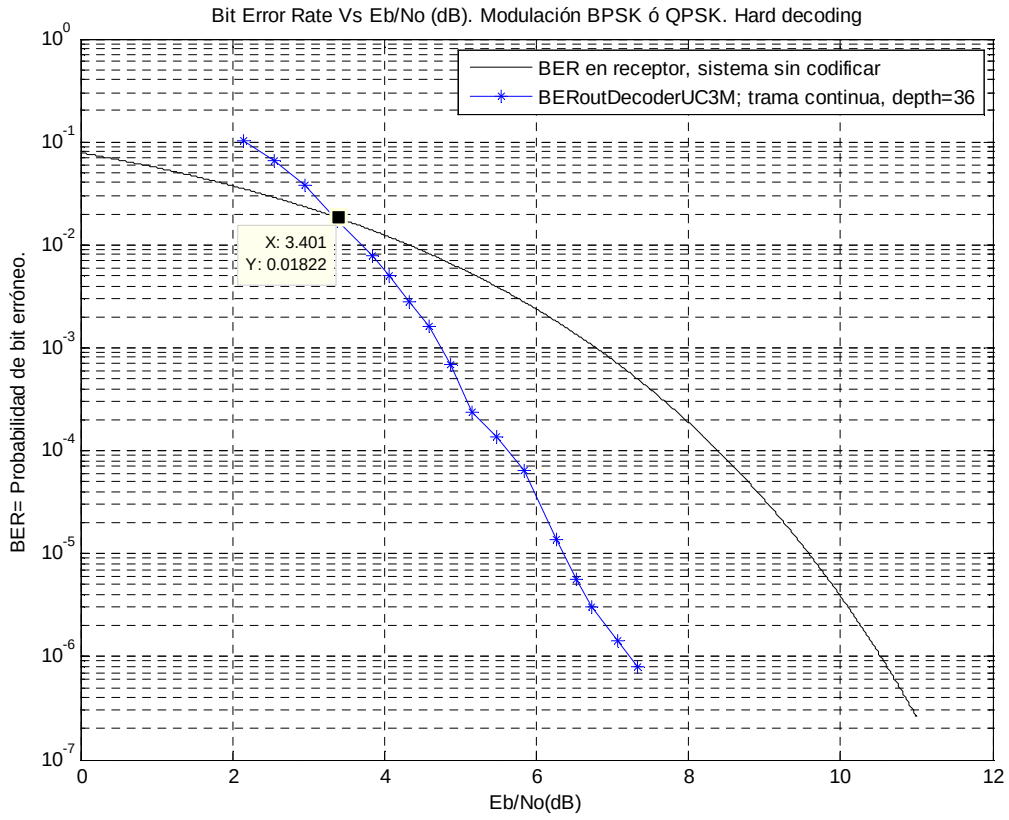


Figura 6.7: Ejemplo de representación de la BER sin codificar y BERoutDecoder .

Una de las curvas representa la BER que se obtendría en el receptor en el caso de que no se emplease codificación convolucional En este caso en el receptor hay una única BER.

La otra curva representa la BER que se obtiene en la salida del decodificador, en el caso de que se emplee un sistema con codificación convolucional en el transmisor y decodificación en el receptor. En este caso en el receptor hay dos BERs, la BERinDecoder y la BERoutDecoder.

A continuación explicamos el proceso a seguir para representar las gráficas como la anterior.

En un sistema sin codificar, cada bit de datos m[n], ver *figura 6.1*, se envía directamente. En cambio en un sistema con codificación convolucional se añade redundancia en la transmisión, según la tasa o code rate, R = k/n. De manera que de cada k bits de datos en m[n] se envían n bits en c[n]. Siempre se cumple que k/n <1 y en nuestro caso tenemos k = 1 y n = 2.

Entonces en el sistema con codificación la relación Eb/No en el receptor es más baja, puesto que se deben transmitir más bits empleando la misma potencia. Esto se demuestra en [15], de dónde extraemos las siguientes fórmulas:

En el sistema sin codificar tenemos $\frac{Eb_{SinCodificar}}{No}$.
En el sistema con codificación se mantiene la misma potencia de transmisión y el mismo ruido de manera que tenemos:

$\frac{Eb_{SinCodificar}}{No} = \frac{Es_{ConCodificacion}}{No} = \frac{n}{k}\frac{Eb_{ConCodificacion}}{No}$; Por cada k bits de datos se envía un símbolo de n bits. Siendo Es la energía de ese símbolo.

$$\frac{Eb_{SinCodificar}}{No}(dB) = \frac{Eb_{ConCodificacion}}{No}(dB) + 10\log_{10}\left(\frac{n}{k}\right)$$

Como n/k es siempre mayor que 1, una primera consecuencia de aplicar la codificación es que la relación Eb/No disminuye en la entrada del decodificador.

$$\frac{Eb_{ConCodificacion}}{No}(dB) = \frac{Eb_{SinCodificar}}{No}(dB) - 10\log_{10}\left(\frac{n}{k}\right)\Big|_{\frac{n}{k}=2} = \frac{Eb_{SinCodificar}}{No}(dB) - 3{,}01dB$$

La consecuencia es que la BER en el receptor en el sistema sin codificar es menor que la BERinDecoder en el sistema con codificación. Así que inicialmente la codificación es perjudicial para el sistema. Lo que ocurre es que BERoutDecoder es menor que BERinDecoder. Entonces el objetivo que se busca, es que la ganancia de BERoutDecoder frente a BERinDecoder supere la pérdida $10\log_{10}\left(\frac{n}{k}\right)$, que produce la redundancia que añade el codificador.

En la *figura 6.7* apreciamos que el sistema de codificación-decodificación no siempre interesa. Porque si en nuestro sistema la relación $Eb_{SinCodificar}/No$ es baja, la ganancia que produce el decodificador no será suficiente para compensar la pérdida $10\log_{10}\left(\frac{n}{k}\right)$. En este caso sólo interesa la codificación cuando la $\frac{Eb_{SinCodificar}}{No}$ es mayor de 3,4 dB.

Las gráficas BER frente a EB/No, como *la figura 6.7*, se representan de la siguiente manera:

El eje de abscisas siempre se refiere a $\frac{Eb_{SinCodificar}}{No}$ (dB), pero por simplicidad usaremos la notación $\frac{Eb}{No}$ (dB)

La curva BER en el sistema sin codificar es una representación de la ecuación 2:

Ecuación 2: $BER = \frac{1}{2} erfc\left(\sqrt{\frac{Eb_{SinCodificar}}{No}}\right) = Q\left(\sqrt{\frac{2Eb_{SinCodificar}}{No}}\right)$

El resto de curvas relacionan BERoutDecoder con $\frac{Eb_{SinCodificar}}{No}$ (dB). Para representarlas tenemos las medidas que hemos obtenido con el simulador.

En el simulador tenemos BERinDecoder como parámetro de entrada y BERoutDecoder como parámetro de salida. Pero lo que necesitamos es relacionar BERoutDecoder con $\frac{Eb_{SinCodificar}}{No}$ (dB). Esto se consigue aplicando esta ecuación:

$$BERinDecoder = \frac{1}{2} erfc\left(\sqrt{\frac{Eb_{ConCodificacion}}{No}}\right) = Q\left(\sqrt{\frac{2Eb_{ConCodificacion}}{No}}\right) \Rightarrow$$

Sustituyendo $\frac{Eb_{ConCodificacion}}{No} = \frac{k}{n}\frac{Eb_{SinCodificar}}{No}$

Ecuación 3. $BERinDecoder = \frac{1}{2} erfc\left(\sqrt{\frac{k}{n}\frac{Eb_{SinCodificar}}{No}}\right) = Q\left(\sqrt{\frac{k}{n}\frac{2Eb_{SinCodificar}}{No}}\right)$

Mediante Matlab representamos la ecuación 3, *figura 6.8*, y buscamos los pares de puntos (BERindecoder, $Eb_{SinCodificar}/No$) que necesitamos. Estos puntos están en las columnas 1 y 2 de la *tabla 6.1*. En la figura también hemos representado la BER en el sistema sin codificar, para apreciar como BERinDecoder tiene una pérdida de 3,01 dB frente a BER Esta diferencia entre las dos curvas se mantiene constante para todos los valores de $Eb_{SinCodificar}/No$.

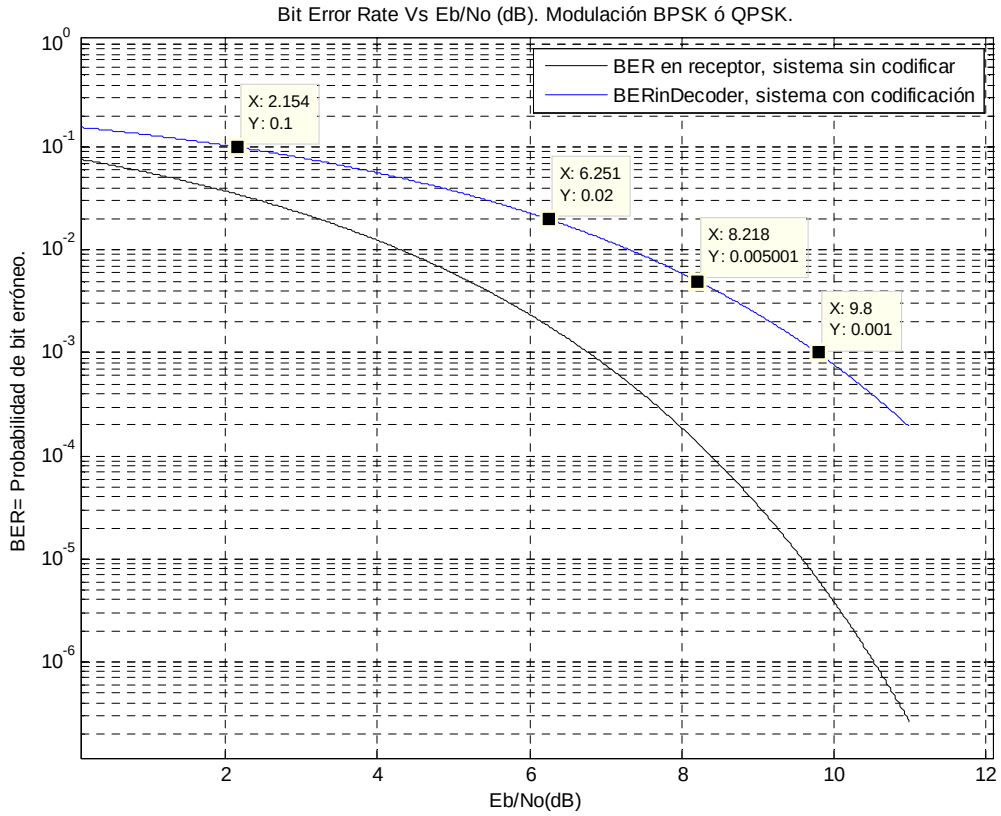


Figura 6.8: Representación ecuación característica BERinDecoder.

Tabla 6.1: Puntos para la representación de BERinDecoder frente a Eb/No		
$Eb_{SinCodificar}$/No dB	BERinDecoder	BerOutDecoderUC3M;Depth = 36, trama continua
2,154	0,1	0.1026
2,5465	0,09	$6{,}441 * 10^{-2}$
2,953	0,08	$3{,}724 * 10^{-2}$
3,38	0,07	$1{,}733 * 10^{-2}$
3,8325	0,06	$7{,}686 * 10^{-3}$
3,072	0,055	$5{,}048 * 10^{-3}$
4,323	0,05	$2{,}803 * 10^{-3}$
4,585	0,045	$1{,}59 * 10^{-3}$
4,864	0,04	$6{,}738 * 10^{-4}$
5,162	0,035	$2{,}387 * 10^{-4}$
5,485	0,03	$1{,}362 * 10^{-4}$
5,845	0,025	$6{,}356 * 10^{-5}$
6,251	0,02	$1{,}37 * 10^{-5}$
6,528	0,017	$5{,}602 * 10^{-6}$
6,731	0,015	$3{,}071 * 10^{-6}$
7,072	0,012	$1{,}426 * 10^{-6}$
7,336	0,01	$8{,}029 * 10^{-7}$
7,809	0,007	Sin medir
8,218	0,005	Sin medir
9,181	0,002	Sin medir
9,8	0,001	Sin medir

Con los datos de la tabla ya podemos dibujar las gráficas, para ello hay que seguir estos pasos:

1. Se ejecuta varias veces el simulador, variando BERinDecoder en cada ejecución. Cuando BERoutDecoder converja rellenamos la columna 3 de la tabla anterior. *Nota 6.1*: BERoutDecoder se expresa en el simulador en %, pero en la tabla y en las gráficas pondremos siempre su valor real entre 0 y 1.

2. La gráfica sin codificar se obtiene representando la *ecuación 2*.

3. La gráfica BERoutDecoder la constituyen los pares de puntos de las columnas 1 y 3. Aparecen algunos puntos sin medir porque la BER es tan baja que se necesitan muchos millones de bits en la simulación para que la BER converja. Esto supondría mucho tiempo de computación, más de 48 horas por cada medida.

4. Los valores de la columna 1 son independientes del decodificador, porque están medidos en su entrada. En cambio los de la columna 3 varían al cambiar los parámetros del decodificador, ya que están tomados en su salida.

6.5 Arquitectura simuladores en VHDL.

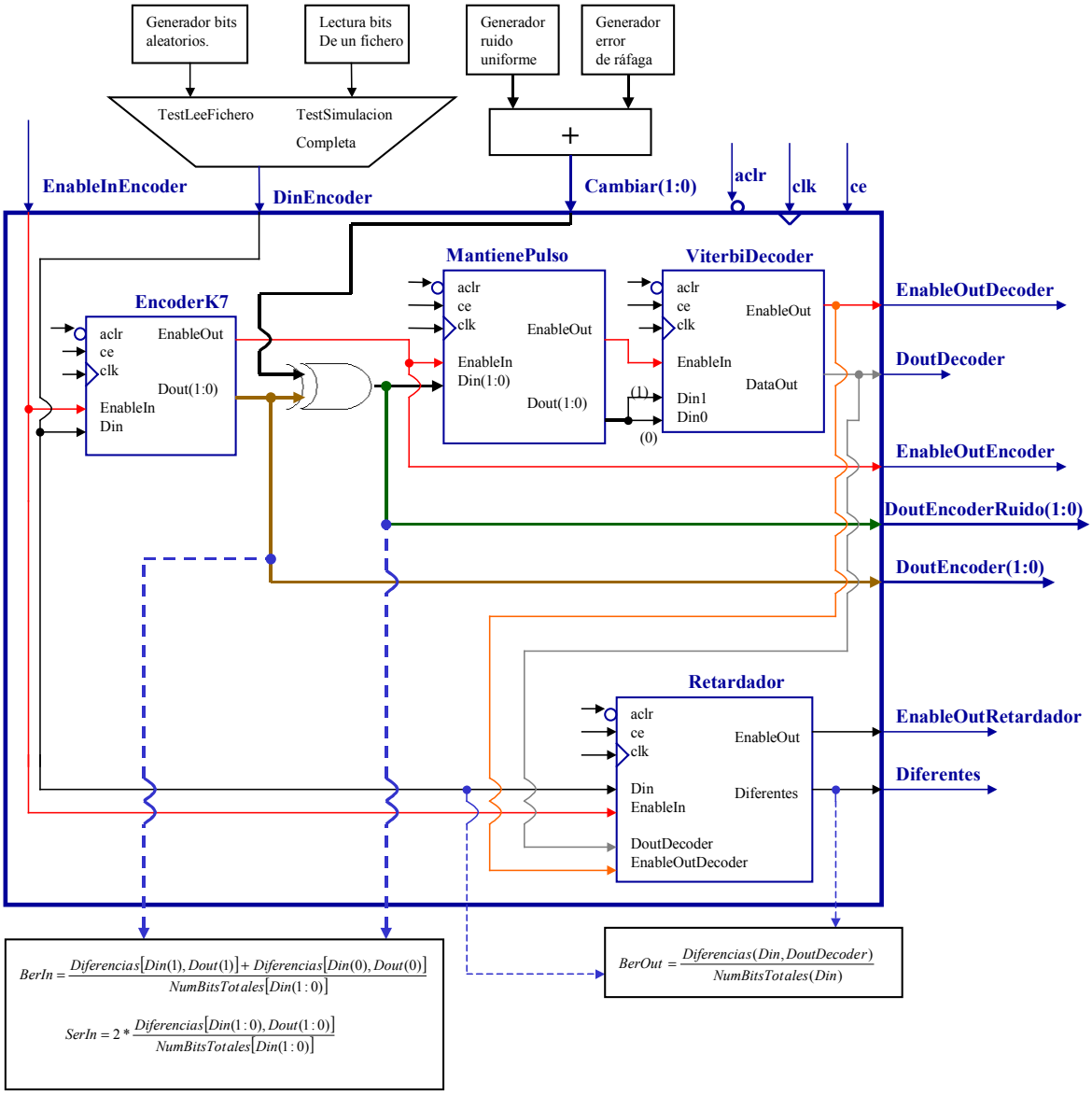


Figura 6.9. Arquitectura simuladores VHDL.

Consiste en codificar el sistema de la *figura 6.2*. El proceso es:

1. Se obtiene una cadena de bits aleatorios. Mediante un generador pseudoaleatorio en el modelo *TestSimulacionCompleta.vhd*, o leyendo la cadena desde un fichero de entrada en *TestLeeFichero.vhd*.

2. EncoderK7 es el codificador convolucional que hemos desarrollado: *EncoderK7.vhd*, *ver tema 3.*

3. Se obtiene el ruido uniforme y el de ráfaga mediante generadores pseudoaleatorios, y se suma a la cadena de bits mediante la puerta xor.

4. El bloque MantienePulso adapta las señales para que sean compatibles con la entrada al *ViterbiDecoder.vhd.*

5. El bloque Retardador añade el retardo necesario a la cadena de entrada al codificador, para poder compararla con la salida del decodificador.

Nota 6.2: Sólo incluye el decodificador UC3M u Opencores. La inclusión de dos decodificadores al mismo tiempo sólo la hemos implementado en los modelos de System Generator.

Nota 6.3: La arquitectura para simular el decodificador de Opencores es la misma. Sólo hay que cambiar *ViterbiDecoder.vhd* por *decoderverilog.v*.

6.5.1 MantienePulso.vhd.

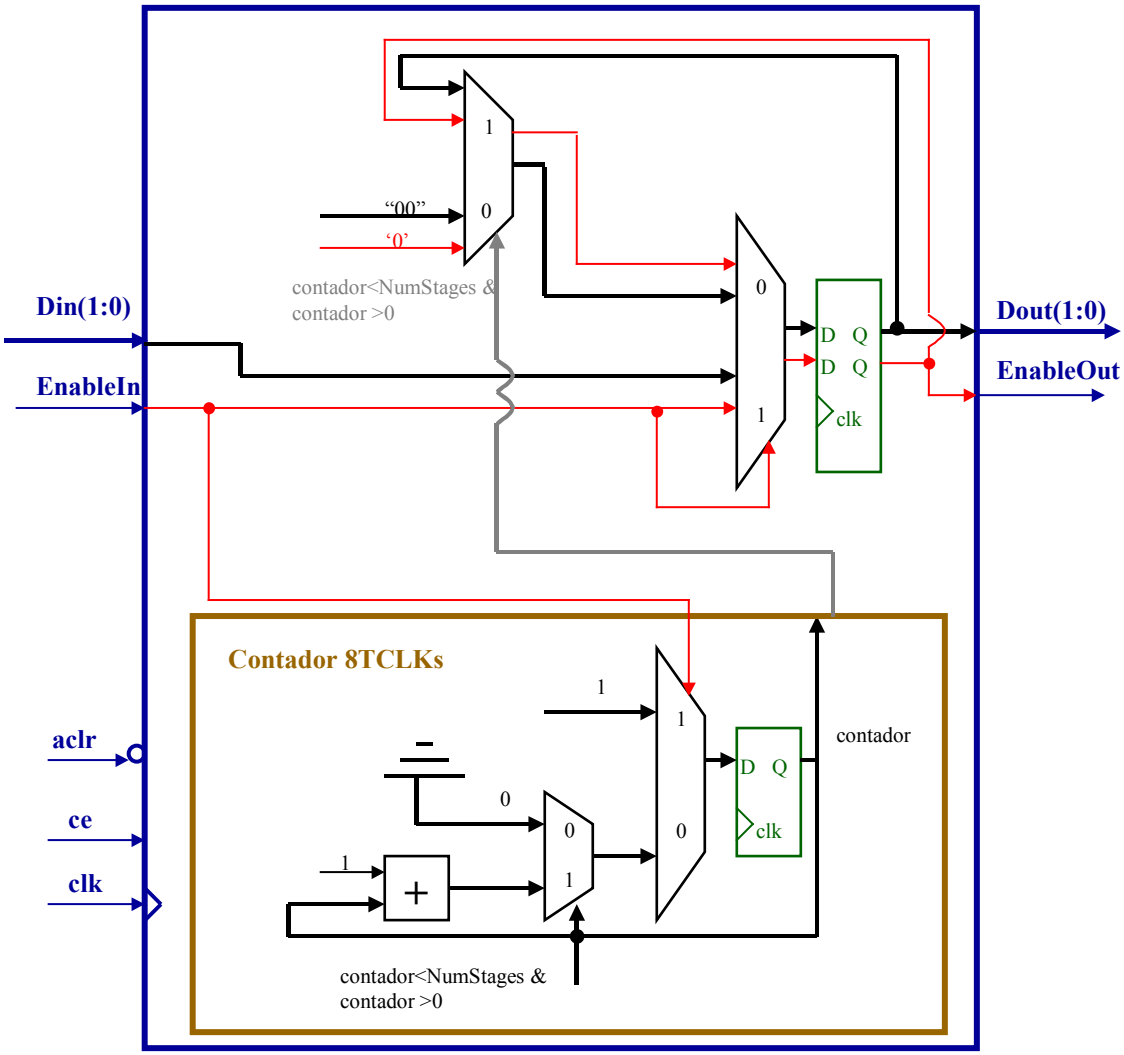
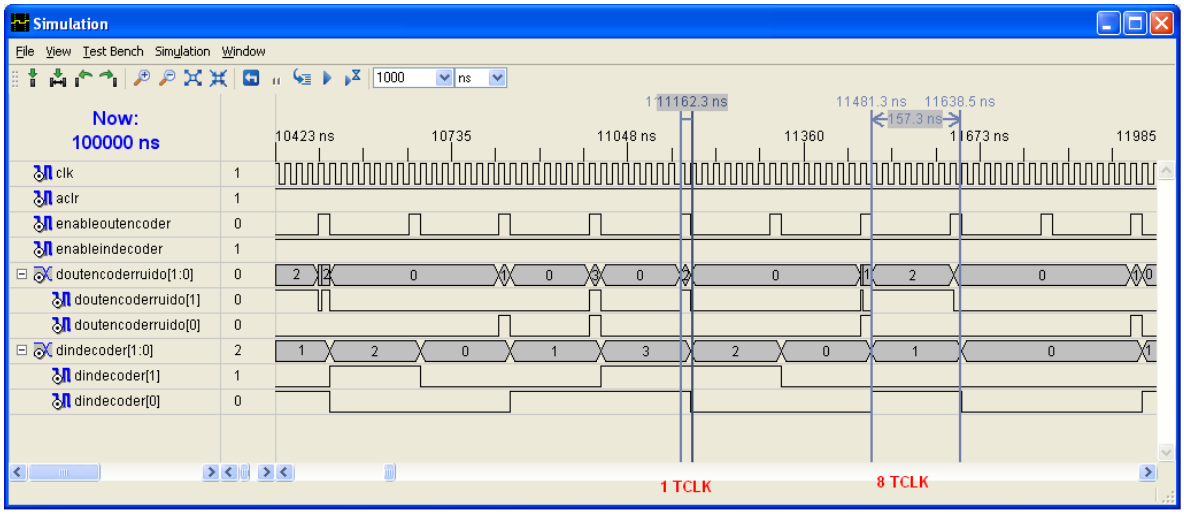


Figura 6.10: MantienePulso.vhd.

Latencia 1 T_{CLK}.

El período de proceso de un dato en el codificador es de 1 T_{CLK} y en el decodificador de 8 T_{CLKs}. Mediante este módulo adaptamos la salida del codificador a la entrada del decodificador para poder trabajar con una única frecuencia de reloj en el simulador.



Cronograma 6.1: MantienePulso.vhd

EnableInDecoder debe estar activo durante todo el período de proceso, 8 T_{CLKs}. Y el dato en DinDecoder también debe estar estable durante 8 T_{CLKs}. Sin embargo EnableOutEncoder es un pulso de 1 T_{CLK}, y está a cero durante los restantes 7 T_{CLKs} del período de proceso del decodificador. Lo mismo sucede con DoutEncoderRuido, que sólo está disponible durante 1 T_{CLK}.

Este bloque alarga la duración del pulso de EnableOutEncoder y DoutEncoderRuido hasta los 8 T_{CLKs}, para que sean compatibles con EnableInDecoder y DinDecoder.

6.5.2 Retardador.vhd.

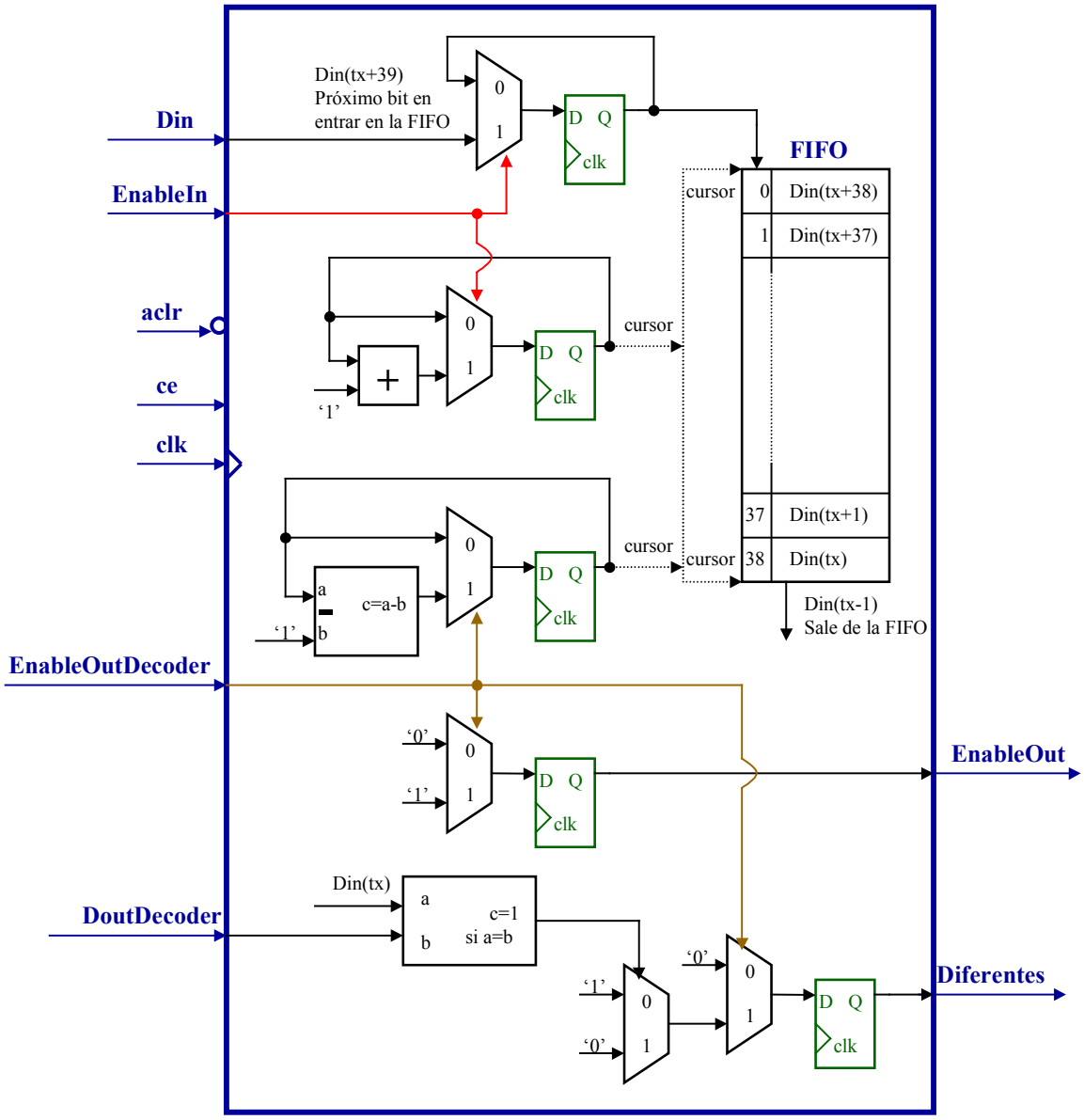


Figura 6.11: Retardador.vhd para decoding depth = 36

Aplica un retardo igual a la latencia del sistema a los bits de entrada al codificador, para compararlos con los de salida del decodificador.

Sirve para calcular $BERoutDecoder = \frac{\text{Nº de veces que diferentes está a '1'}}{\text{Nº total de bits decodificados}}$

El retardo del sistema es igual a la latencia entre EnableInEncoder y EnableOutDecoder. Su valor es la suma de todas las latencias del sistema:

- EncoderK7 → 1 T_{CLK}.
- MantienePulso → 1 T_{CLK}.
- ViterbiDecoder → 10 + (decoding depth+1)*8 T_{CLKs}

El valor depende de decoding depth, además en los simuladores de System Generator se incluye el decodificador de Xilinx, que tiene una latencia diferente.

Tabla 6.2: Latencia entre EnableInEncoder y EnableOutDecoder					
Nota 6.4: Latencia en períodos de proceso (8 T_{CLKs})	Decoding depth				
Decodificador	24	32	36	48	60
ViterbiDecoder.vhd	26,5	34,5	38,5	50,5	62,5
Decoderverilog.v		36,875			
Viterbi decoder v5.0 (Xilinx)	115		163	211	259

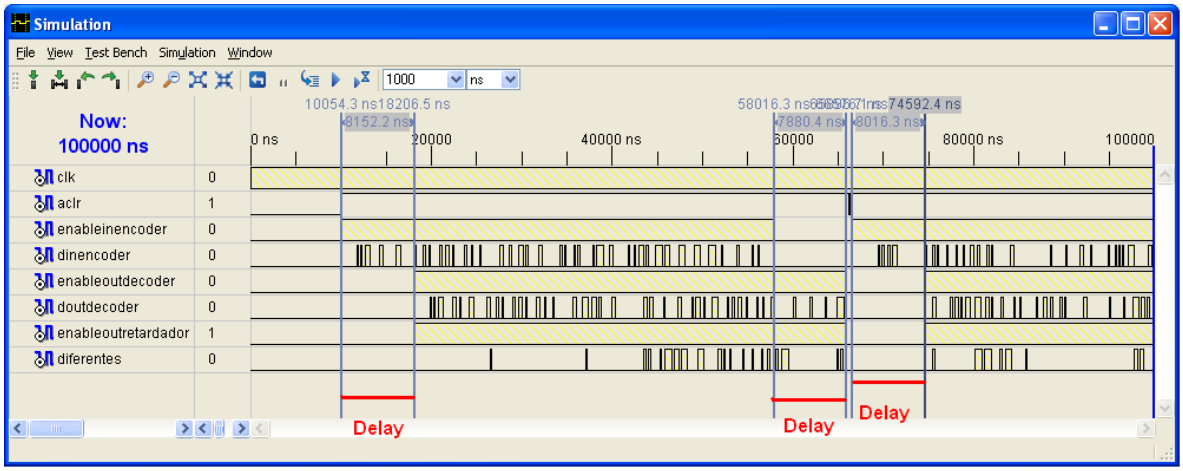
Nota 6.5: Las latencias las obtenemos observando los resultados tras la simulación mediante gráficas similares a la *figura 7.1*.

El funcionamiento del bloque es:

1. Cada bit de entrada al codificador se almacena en la primera posición (0) de una memoria FIFO, y se desplazan una posición los que ya están. La FIFO la hemos diseñado nosotros mismos, en vez de usar un Core Generator de Xilinx, para asegurarnos de que el simulador es totalmente multiplataforma.

2. Una vez alcanzados un número de bits igual a la latencia del sistema, entonces en cada período de proceso, se almacena el bit de entrada al codificador en la posición 0. Y se desplazan una posición los que ya están en la FIFO, de manera que el último sale.

3. El bit que sale se compara con DoutDecoder. Si son iguales, entonces la señal Diferentes valdrá '0'. Pero si no lo son, Diferentes valdrá '1', y esto indica el bit es erróneo porque no es igual al DinEncoder correspondiente



Cronograma 6.2: Retardo en el sistema.

El bloque retardador debe servir para cualquier simulador con cualquier decoding depth, tanto de VHDL como de System Generator. En la *tabla 6.2* vemos que en cada caso hay que aplicar un retardo diferente. Sin embargo le hemos dado un importante valor añadido a este bloque. Consiste en que vale para cualquier decodificador de la *tabla 6.2* sin hacer ningún cambio. El bloque retardador instanciado en cada uno de nuestros simuladores es exactamente igual en todos.

Para ello sólo hay que utilizar un valor alto en la constante RetardoMax. Esta constante determina el número máximo de períodos de proceso de retardo que puede aplicar el bloque retardador. Entonces le damos un valor mayor que 259, ejemplo 400, y así nos vale para todo nuestro diseño sin tener que volver a tocar esa constante nunca.

También es importante indicar que utilizar un valor alto no penaliza en nada al simulador. Entonces no tiene sentido utilizar un valor ajustado a cada decoding depth e ir cambiándolo. El área del simulador sí aumenta, pero esto no afecta al diseño, puesto que en la placa sólo irá el decodificador. El resto de bloques del simulador no se sintetizarán en el diseño final.

6.6 Manual de usuario Simuladores en VHDL.

Utilizamos el ISE Simulator de Xilinx, ISim, documentación en [1], [2] y [3].

6.6.1 Resultados

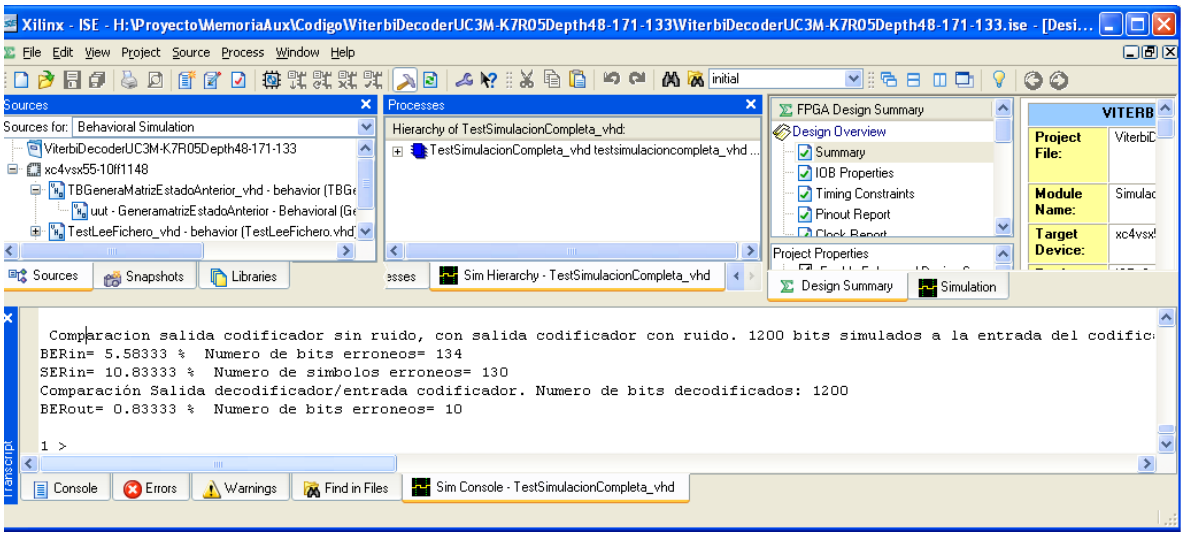


Figura 6.12: Consola del simulador VHDL.

Al finalizar la simulación del último bit, se muestra en la consola toda la información de interés:

1. La primera línea indica el número total de bits que han entrado al codificador durante el transcurso de la simulación. Debemos tener en cuenta que como la tasa es R= 1/2, en la entrada del decodificador habrá el doble de bits.

2. En la segunda línea tenemos la BER en la entrada del decodificador, tras añadirle el ruido. Y el número de bits diferentes entre la salida del codificador y la entrada al decodificador.

$$BER_{in} = \frac{\text{bits erroneos entrada decodificador}}{\text{bits totales entrada decodificador}} * 100 = \frac{134}{2400} * 100 = 5{,}5833\%$$

3. SER en la entrada del decodificador, tras añadirle el ruido. Y el número de símbolos diferentes entre la salida del codificador y la entrada al decodificador.

$$SER_{in} = \frac{\text{simbolos erroneos entrada decodificador}}{\text{simbolos totales entrada decodificador}} * 100 = \frac{130}{1200} * 100 = 10{,}833\%$$

4. El número de bits decodificados al finalizar la simulación debe coincidir exactamente con el número total de bits en la entrada del codificador.

5. La última línea muestra la información más importante, la BER en la salida del decodificador y el número de bits diferentes entre la entrada al codificador y la salida del decodificador.

$$BER_{out} = \frac{\text{bits erroneos salida decodificador}}{\text{bits totales salida decodificador}} * 100 = \frac{10}{1200} * 100 = 0{,}833\%$$

El simulador también nos sirve como herramienta para depurar el código del decodificador y de todo el sistema de comunicaciones. Para ello nos sirve de mucha ayuda la representación de las señales del sistema, como muestra la siguiente figura:

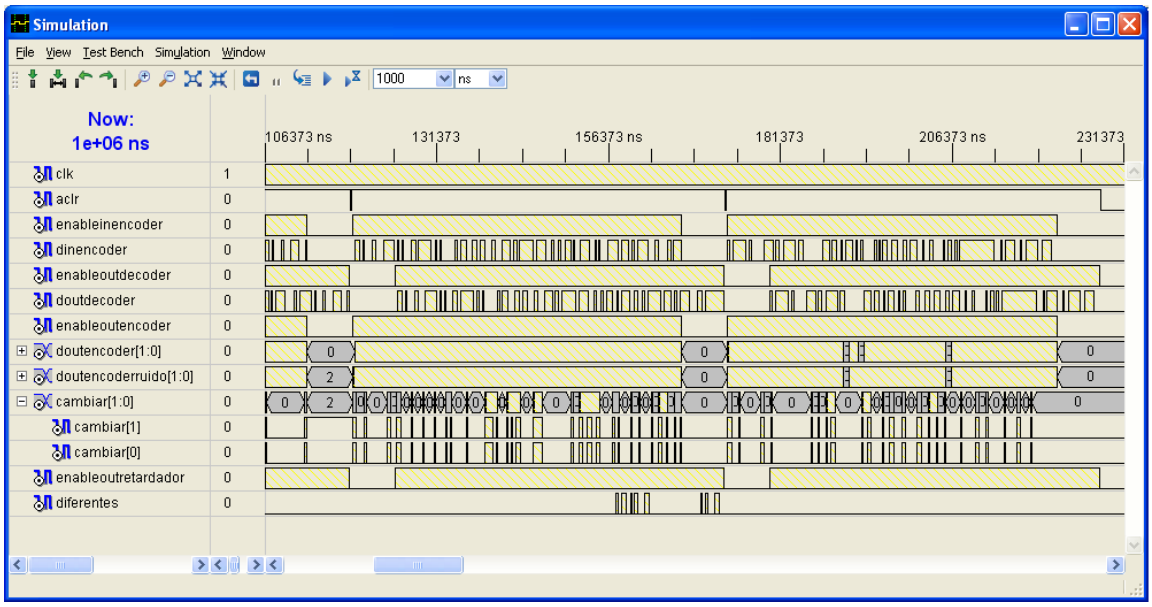


Figura 6.13: Señales del simulador VHDL.

Estudiando las señales anteriores obtenemos esta información

1. La latencia de todo el sistema entre la entrada al codificador y la salida del decodificador.

2. La latencia del decodificador.

3. El tiempo de espera necesario entre el final de una trama de entrada y el comienzo de la siguiente.

4. Si la activación de $\overline{aclr}$ entre trama y trama se realiza en el momento adecuado.

5. Cada pulso en Cambiar1 y Cambiar0 representa un bit modificado por el ruido en la entrada del decodificador y cada pulso en Diferentes representa un bit decodificado erróneo. Vemos que el número de pulsos en Diferentes es menor que en Cambiar(1:0), por lo que el decodificador funciona correctamente y disminuye el número de errores.

6.6.2 Configuración simulador.

Tiempo de simulación

Debemos especificar durante cuanto tiempo se ejecutará el comando run.

- TestLeeFichero.vhd →

Tsimulacion = [500 + 8*Numero bits a simular + 550]*T_{CLK}

- *TesSimulacionCompleta.vhd*

TSimulacion = [500 + 8*Numero bits a simular + EsperaTramasMin*8*(Num tramas-1) + 550]*T_{CLK}

Los parámetros por defecto son:

1. T_{CLK} = 20 ns.

2. 500 corresponde al tiempo en el que $\overline{aclr}$ está activa: constante NumCiclosReset = 500.

3. Cada bit necesita un período de proceso, 8 T_{CLKs} para decodificarse.

4. EsperaTramasMin = decoding depth +4.

5. 550 es el tiempo de margen que hay que dejar al simulador para que termine la decodificación de los bits, una vez que haya llegado el último bit a la entrada del codificador. Constant esperaMax = 550;

Ficheros de entrada

En *TestLeeFichero.vhd* tenemos **BitsSimulacion.txt**. Contiene todos los bits de entrada al codificador, un bit en cada línea. Se genera ejecutando *GenFichAleatorio.m*. Sólo hay bits de datos, no debe haber una cadena inicial ni final de ceros. El decodificador usa la técnica truncamiento de trellis.

En *TestSimulacionCompleta.vhd* no hay ficheros de entrada, la cadena de bits a simular la obtiene el propio simulador mediante un generador pseudoaleatorio.

Variación decoding depth.

En *TestLeefichero.vhd* no hay que modificar nada.

En *TestSimulacionCompleta.vhd* hay que asignar el valor correcto a **EsperaTramasMin**, que debe ser ≥ decoding depth + 4. Indica el número de períodos de proceso de un dato que EnableInEncoder permanece inactivo entre dos tramas consecutivas.

Constantes a modificar antes de lanzar la simulación.

BER: Cantidad de ruido uniforme en la entrada del decodificador. 0,0 ≤ BER ≤ 1,0 . Este valor no se expresa en %. Sin embargo debemos tener en cuenta que en los resultados se da BERin, SERin y BERout en %.

NumBitsSimulacion y **NumBitsTrama**.
Con estas dos constantes se especifica el número de bits a simular y en cuántas tramas irán organizados. NumBitsSimulacion = X*NumBitsTrama, con X entero.
Sólo está disponible en *TestSimulacionCompleta.vhd.*

Initial 1 y 2: Semillas del generador de entrada del codificador. Sólo en *TestSimulacionCompleta.vhd.*

Initial 3..6 : Semillas del generador de ruido. No se modifica la BER, lo que cambia es la distribución de los bits erróneos.

ErrorRafaga: Si vale '1', entonces la simulación es con errores de ráfaga. Si vale cero sin errores de ráfaga.

Modificar sólo si ErrorRafaga = '1':

- **MaxIntervaloRafaga** y **MinIntervaloRafaga**. Intervalo en bits entre dos ráfagas consecutivas.

- **MaxLongRafaga** y **MinLongRafaga**. Duración en bits de una ráfaga.

- **Initial 7..10**. En caso de que los intervalos entre ráfagas y su duración no sean fijos, con estas semillas se modifica la generación de esas distancias.

Ficheros de salida

BitsInEncoder.txt : Los bits aleatorios generados que constituyen la entrada al codificador. Sólo en *TestSimulacionCompleta.vhd*. En *TestLeeFichero.vhd* no existe este fichero de salida, porque los bits de entrada al codificador se leen del fichero de entrada *BitsSimulacion.txt*.

El resto de ficheros están presentes tanto en *TestLeeFichero.vhd* como en *TestSimulacionCompleta.vhd.*

BitsOutEncoder.txt: Los bits tras pasar por el codificador convolucional, y antes de añadirles el ruido. 2 bits, un símbolo por línea.

BitsOutEncoderRuido.txt : Los bits codificados tras añadirles el ruido uniforme y los errores de ráfaga. Constituyen la entrada al decodificador, 2 bits, un símbolo por línea.

BitsOutDecoder.txt: Los bits de salida del decodificador, 1 bit por línea.

Comparando los ficheros se calcula la BERin y BERout del sistema.

$$BERin = \frac{\text{número de bits diferentes entre BitsOutEncoder y BitsOutEncoderRuido}}{\text{numero de bits en BitsOutEncoder}}$$

$$SERin = \frac{\text{número de líneas diferentes entre BitsOutEncoder y BitsOutEncoderRuido}}{\text{numero de líneas en BitsOutEncoder}}$$

$$BERout = \frac{\text{número de diferencias entre (BitsInEncoder o BitsSimulacion) y BitsOutDecoder}}{\text{numero de bits en BitsOutDecoder}}$$

El cálculo de las diferencias es automático mediante *ComparaFicheros.m.*

En las fases iniciales del diseño, no habíamos desarrollado la aplicación que calcula automáticamente BERin, SERin y BERout y muestra el resultado en la consola. El proceso lo hacíamos manualmente comparando los ficheros de salida mediante la función de Matlab *ComparaFicheros.m.*.

Por tanto en el simulador final estos archivos no nos resultan demasiado útiles. Sin embargo los mantenemos porque su generación no supone ninguna desventaja en el funcionamiento del simulador.

6.7 Arquitectura simuladores en System Generator.

Documentación sobre System Generator en el *apartado 1.9.2*.

La arquitectura es la misma que en la *figura 6.9*, realizando los cambios necesarios para adaptarla a la nueva herramienta.

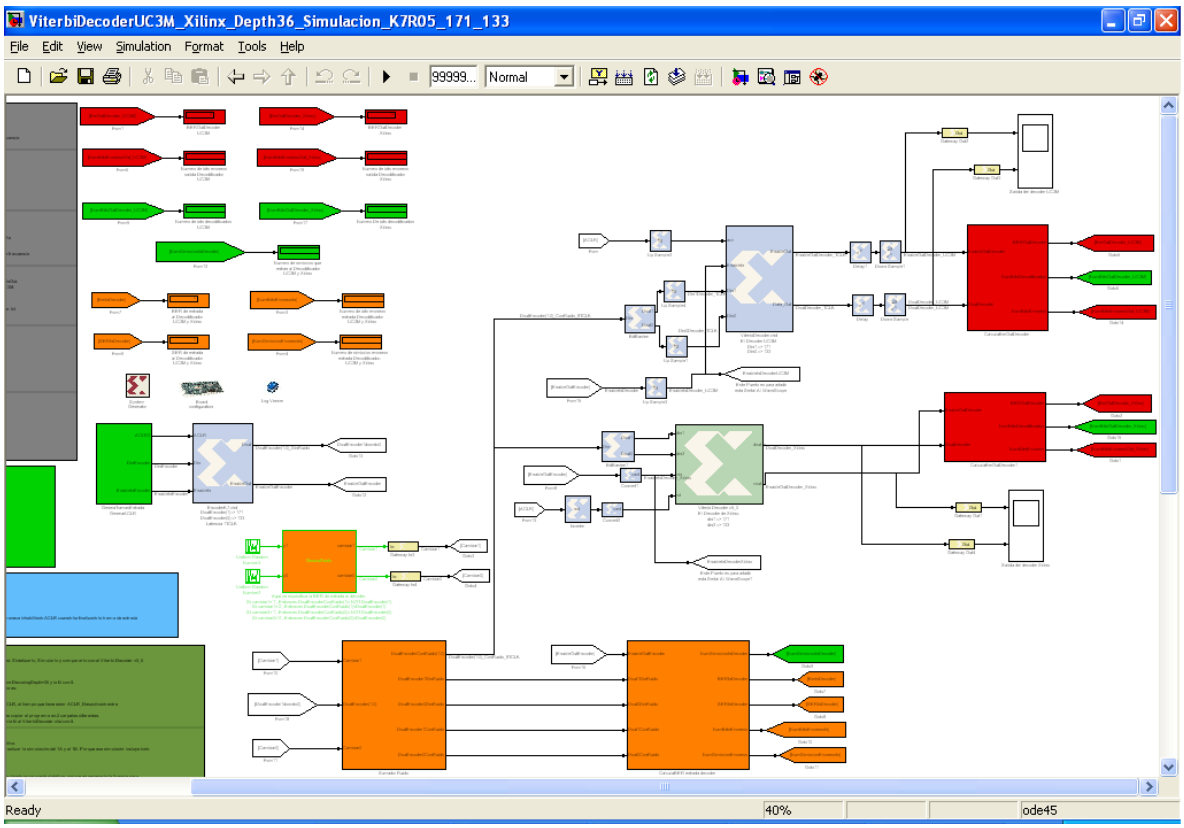


Figura 6.14. Arquitectura simuladores System Generator.

La figura anterior muestra los bloques que constituyen el simulador. Se trata únicamente de los principales, puesto que dentro de ellos hay múltiples subbloques.

En esta memoria no trataremos en ningún momento de explicar detalladamente los bloques, porque en el código del simulador hemos añadido múltiples comentarios que explican exhaustivamente su arquitectura y funcionalidad. De manera que no tiene sentido abarcar la explicación de nuevo en esta memoria. Por tanto, la mejor forma de comprender los detalles del funcionamiento, es leyendo detenidamente esos comentarios y navegando por los subbloques del simulador.

El objetivo y el funcionamiento es igual que en los simuladores que hemos realizado en VHDL. La descripción en el *apartado 6.5* es también válida para explicar la arquitectura de los simuladores de System Generator. Únicamente debemos tener en cuenta que hay que adaptar el código a la nueva herramienta.

6.7.1 Archivos implementados.

En primer lugar importamos de VHDL los archivos que implementamos para realizar el simulador VHDL.

- EncoderK7.vhd
- ViterbiDecoder.vhd
- Decoderverilog.v
- Retardador.vhd
- MantienePulso.vhd

Los bloques anteriores son la base del simulador, pero no son suficientes. Debemos implementar nuevos bloques y funciones, para conseguir la misma funcionalidad que obteníamos con los simuladores de VHDL. Estos añadidos son:

- GeneraTramasEntradaGeneraACLR
  - GeneraACLR_T1.m. Controla la activación y desactivación de $\overline{aclr}$.
  - GeneraACLR_T8.m.
  - Y. Fija el numero de bits de cada trama de entrada.

- GeneraRuido.

- SumadorRuido.

- CalculaBER entrada decoder.
  - CalculaBERinDecoder.m.

- Viterbi decoder v5_0. Es el IP core de Xilinx que implementa un decodificador Viterbi. Datasheet en [16A].

- CalculaBEROutDecoder.
  - Retardador.vhd.
  - CalculaBEROutDecoder.m.

6.7.2 Añadidos para adaptarse a la técnica cola de ceros.

Los decodificadores que hemos desarrollado, el UC3M y el de Opencores, emplean la técnica truncamiento de trellis. Pero el IP core de Xilinx utiliza cola de ceros, [17] y [18]. Esto significa que en nuestros decodificadores, no hace falta insertar bits de relleno ni al principio ni al final de la trama de datos de entrada. En cambio en el de Xilinx, es necesario añadir una cola de 6 ceros al final de cada una de las tramas.

Por eso en el simulador que compara el decodificador de Xilinx con el UC3M, hemos modificado el bloque GeneraTramasEntradaGeneraACLR, añadiendo 6 ceros al final de cada trama de datos. Las tramas de datos más los 6 ceros al final constituyen la entrada de los dos decodificadores. Añadimos los ceros de relleno también al *ViterbiDecoder.vhd*, para que así los símbolos de entrada sean exactamente iguales en los dos decodificadores.

De manera que si la trama de entrada tiene Y bits de datos. Al decodificador de Xilinx llegarán Y símbolos de datos por trama, los últimos 6 son sólo relleno.

En cambio al *ViterbiDecoder.vhd* llegan Y+6 símbolos de datos, porque interpreta los 6 ceros finales como bits de datos.

6.7.3 Descripción funcionamiento.

Paso 1: El sistema tiene tres señales de entrada, que se generan mediante el bloque GeneraTramasEntradaGeneraACLR. Estas señales son:

a. DinEncoder: Los bits de entrada al codificador, se generan mediante un generador pseudoaleatorio.
b. EnableInEncoder.
c. $\overline{\text{aclr}}$.

Los bits de entrada se organizan mediante una cadena de X tramas de Y bits cada una. También se puede seleccionar una entrada continua, de manera que sólo haya una trama.

En el simulador: ViterbiDecoderUC3M_Depth36_Simulacion_K7R05_171_133, EnableInEncoder está activo durante 1 T_{CLK} y desactivado durante 7 T_{CLKs}. Esto se debe a que en el simulador se trabaja con una única frecuencia $F=1/T_{CLK}$. De manera que el período de proceso del decodificador es de 8 T_{CLKs} y el del codificador de 1 T_{CLK}.

Sin embargo en las versiones posteriores:
ViterbiDecoderUC3M_Depth36_8TCLK_TCLK_K7R05_171_133.mdl
ViterbiDecoderUC3M_Xilinx_Depth36_Simulacion_K7R05_171_133.mdl.

EnableInEncoder está siempre activo. Esto se debe a que en estos simuladores se trabaja con dos frecuencias de reloj simultáneamente. $F_2=1/T_2=1/(8T_{CLK})$ y $F_1=1/T_1=1/T_{CLK}$.
El decodificador UC3M trabaja con la frecuencia $F_1=8F_2$. De esta manera en un sólo período T_2 es capaz de decodificar un dato. Y el codificador trabaja con frecuencia F_2.

Paso 2: Los bits de entrada se codifican mediante EncoderK7.vhd.

Paso 3: Mediante la función GeneraRuido se modela el canal de transmisión. La función obtiene ruido uniforme pseudoaleatorio. El nivel del ruido es variable, mediante la constante BER.
Nota 6.6: En los simuladores de System Generator no se pueden añadir errores de ráfaga.

Paso 4: Se añade el ruido a los bits codificados convolucionalmente. Para ello disponemos del bloque SumadorRuido.

Paso 5: Se calcula la BER en la entrada del decodificador y se muestra el resultado instantáneo en la consola.

Paso 6: Los bits codificados más el ruido son la entrada al ó a los decodificadores Viterbi. Hay que adaptar las diferencias entre el período de proceso del codificador y el del decodificador. Para ello utilizaremos el bloque *MantienePulso.vhd* en la versión ViterbiDecoderUC3M_Depth36_Simulacion_K7R05_171_133.
Y bloques Up Sample y Down Sample en las dos versiones posteriores, en las que ya se trabaja con 2 frecuencias de reloj al mismo tiempo.

Paso 7: Mediante CalculaBerOutDecoder se obtiene la BER en la salida del decodificador y se muestra el resultado instantáneo en la consola. En el caso de que se simulen 2 decodificadores al mismo tiempo, se calcula la BER en la salida de cada uno de ellos y se muestran ambos resultados.

6.8 Manual de usuario Simuladores en System Generator.

6.8.1 Resultados

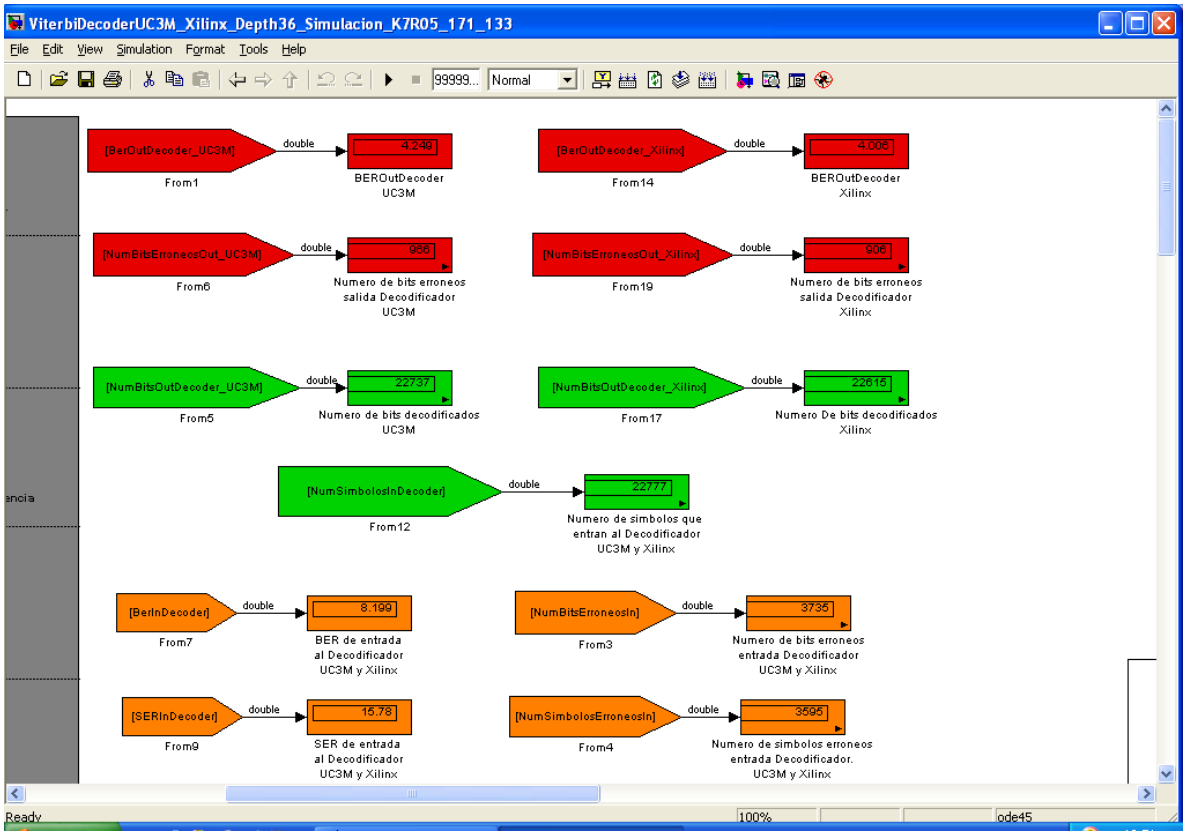


Figura 6.15: Consola del simulador System Generator.

Durante el transcurso de la simulación, los displays se actualizan cada vez que haya un cambio en la variable asociada al display. Gracias a esta característica, los displays se actualizan instántaneamente, mostrando los resultados de la simulación en tiempo real. Esto es una ventaja frente a los simuladores en VHDL, porque en ellos, los resultados de la simulación sólo aparecen en la consola cuando ha finalizado la simulación del último bit.

NumSimbolosInDecoder: Sólo hay un display porque a los dos decodificadores llegan los mismos símbolos de entrada y además lo hacen en el mismo instante. Por este motivo tanto la BERin como la SERin en los dos decodificadores es la misma.

$$BER_{in} = \frac{NumBitsErroneosIn}{2 * NumSimbolosInDecoder} * 100 = \frac{3735}{2 * 22737} * 100 = 8{,}199\,\%$$

$$SER_{in} = \frac{NumSimbolosErroneosIn}{NumSimbolosInDecoder} * 100 = \frac{3595}{22737} * 100 = 15{,}78\,\%$$

En los simuladores en los que no se instancia el decodificador de Xilinx, el número de símbolos de entrada es igual a (Número de tramas) * (Número de bits de cada trama). Pero en los simuladores en los que se instancia el decodificador de Xilinx, tenemos NumSimbolosInDecoder = (Número de tramas) * (Número de bits de cada trama + 6).

NumBitsOutDecoder: Cuando finaliza el tiempo de simulación debe cumplirse obligatoriamente:
NumBitsOutDecoder_UC3M= NumBitsOutDecoder_Xilinx = NumSimbolosInDecoder
Porque a cada símbolo de entrada le corresponde un bit decodificado en la salida.

Pero antes de finalizar la simulación tenemos:
NumSimbolosInDecoder >NumBitsOutDecoder_UC3M > NumBitsOutDecoder_Xilinx
Porque debido a la latencia del decodificador, los bits decodificados están disponibles en la salida con un retardo respecto a los símbolos de entrada.

Además tenemos NumBitsOutDecoder_UC3M > NumBitsOutDecoder_Xilinx porque la latencia del decodificador de Xilinx es mayor que la del UC3M.

$$BER_{out\_UC3M} = \frac{NumBitsErroneosOut\_UC3M}{NumBitsOutDecoder\_UC3M} * 100 = \frac{966}{22737} * 100 = 4{,}249\,\%$$

$$BER_{out\_Xilinx} = \frac{NumBitsErroneosOut\_Xilinx}{NumBitsOutDecoder\_Xilinx} * 100 = \frac{906}{22615} * 100 = 4{,}006\,\%$$

Nota 6.7: Ni el número de errores en la salida de los dos decodificadores, ni la BERout tienen porqué ser iguales. Es totalmente normal que haya diferencias en ambos valores, y esto no significa que el *ViterbiDecoder.vhd* esté mal diseñado. En el *apartado* 7.7.1 explicamos los motivos. Las diferencias existirán tanto en el valor instantáneo como en el definitivo tras finalizar la simulación.

El simulador también nos sirve como herramienta para estudiar las latencias de los bloque del sistema. Y para depurar el código del decodificador y de todo el sistema de comunicaciones. Aunque en nuestro caso concreto, cuando implementamos estos simuladores, ya habíamos verificado el código del decodificador mediante la simulación en VHDL.

Para ello representamos las señales del sistema, añadiendo un bloque Wave Scope, [16B].

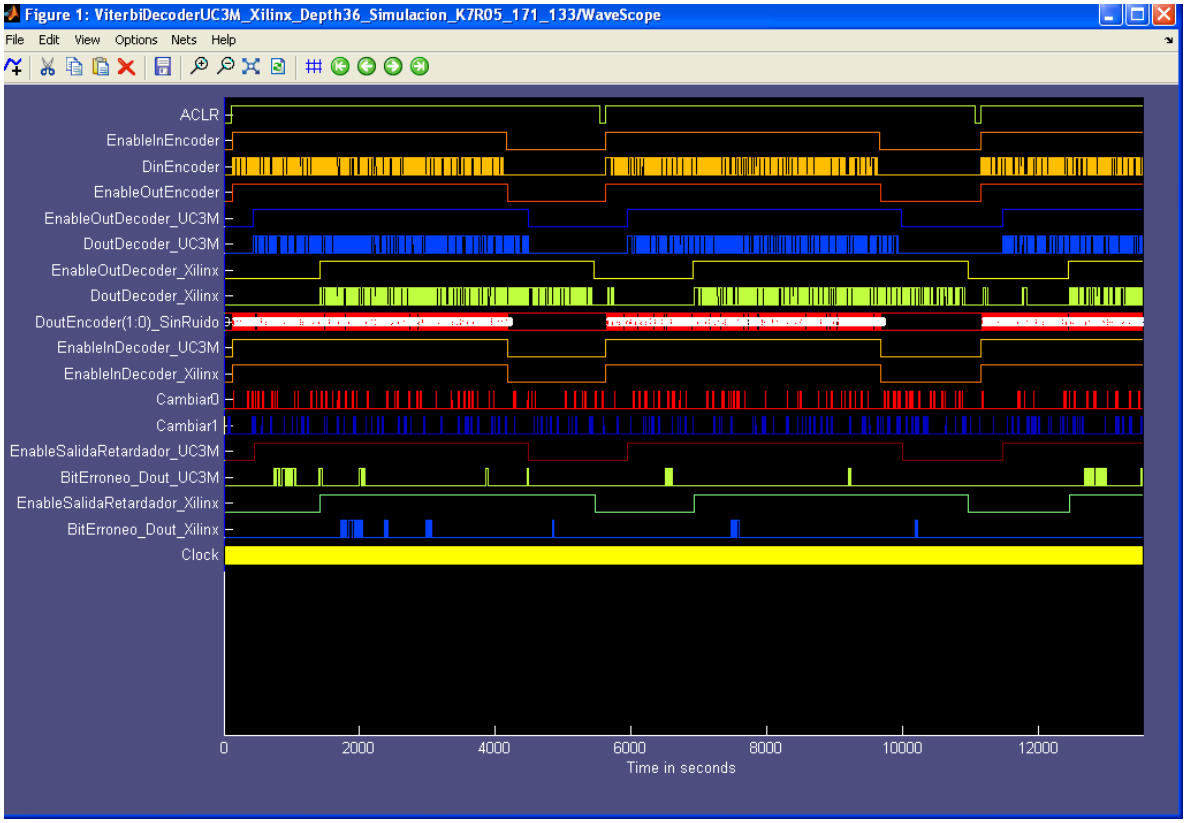


Figura 6.16: Señales del simulador System Generator.

El estudio de estas señales es semejante al que explicábamos tras la *figura 6.13*.

En este caso cada pulso en las señales Cambiar0 y Cambiar1, representa los bits modificados por el ruido en la entrada del decodificador. Y en las señales BitErroneo_Dout representan los errores en la salida. Al igual que en el caso del simulador VHDL, el número de pulsos en la salida es menor que en la entrada, lo que indica que los errores producidos por el ruido han disminuido.

6.8.2 Configuración simulador.

Variación decoding depth.

Los simuladores que hemos realizado funcionan para cualquier decoding depth. El único cambio necesario en su estructura es instanciar el *ViterbiDecoder.vhd* con el decoding depth adecuado. Y modificarlo también en el Viterbi decoder v5_0, [16A].

Sin embargo para simplificar el proceso de simulación al máximo, hemos desarrollado simuladores para cada uno de los decoding depths. Y cada uno de ellos ya instancia el decodificador adecuado.

Pero hay que tener en cuenta que si se divide la simulación en tramas, el decoding depth influye en el simulador, porque se modifica la latencia de los decodificadores. Como se muestra en la siguiente figura:

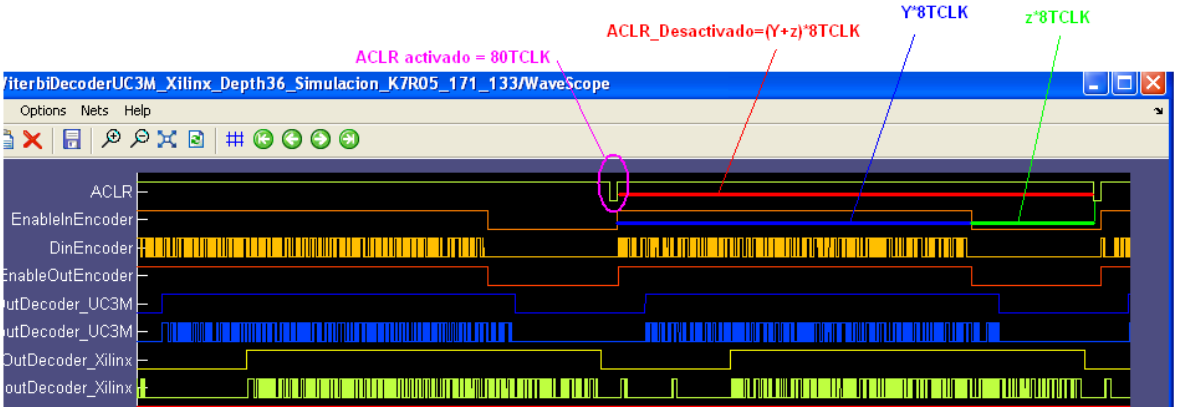


Figura 6.17: Activación y desactivación de aclr entre tramas consecutivas.

Durante una trama, $\overline{aclr}$ debe permanecer inactivo durante al menos Y + z períodos de proceso de un dato. Después es obligatorio activarlo durante al menos 10 períodos de proceso para poner el decodificador en el estado inicial. A continuación, cuando llegue la siguiente trama, se desactiva de nuevo $\overline{aclr}$ para que el decodificador trabaje.

Y es el número de bits de la trama. Z es la latencia del sistema entre EnableInEncoder y EnableOutDecoder. Es necesario sumar este tiempo porque cuando finalizan los bits de la trama de entrada, el decodificador continúa trabajando durante latencia períodos de proceso. Para así decodificar los bits que aún tenga almacenados en su memoria.

Al simular el *ViterbiDecoder.vhd* la latencia de todo el sistema es de decoding depth + 2,25 períodos de proceso.
En el decodificador de Xilinx la latencia de todo el sistema es:
Arquitectura paralelo, optimización área = 4*(decoding depth) + 17
Arquitectura paralelo, optimización velocidad = 4*(decoding depth) + 18

Todo el proceso lo controlamos de una manera muy simple, con una única constante, ACLR_Desactivado, situada dentro del bloque" GeneraTramasEntrada GeneraACLR". Esta constante indica el número de períodos de proceso en los que $\overline{aclr}$ permanece desactivado para procesar una trama.

Debe cumplirse ACLR_Desactivado ≥ Y + z. A continuación se muestran los valores mínimos necesarios para cada uno de los diseños. Se puede poner un valor mayor, pero esto aumenta el tiempo de simulación.

Tabla 6.3: Valor mínimo en ACLR_Desactivado						
	Decoding depth					
	8	24	32	36	48	60
ViterbiDecoderUC3M_Xilinx	Y+80	Y+135		Y+180	Y+230	Y+280
ViterbiDecoderUC3M_8TCLK_TCLK	Y+17	Y+33		Y+45	Y+57	Y+69
ViterbiDecoderUC3M_Simulacion	Y+12	Y+28		Y+40	Y+52	Y+64
ViterbiDecoderUC3M_Opencores			Y+40			

Tiempo de simulación

TSimulacion = $120*T_{CLK}$ +
$8*[NumTramas*(NumBitsTrama + 6) + (ACLR\_Desactivado + 10)*(NumTramas)]*T_{CLK}$

- TCLK = 1 segundo.
- 120 Corresponde al tiempo en el que $\overline{aclr}$ está activa antes de iniciarse la simulación.
- Cada bit necesita un período de proceso 8 T_{CLKs} para decodificarse. En el número de bits a simular no se incluyen los 6 ceros añadidos al final de cada trama.
- Entre trama y trama hay que esperar (ACLR_Desactivado+10)*8 T_{CLKs}. ACLR_Desactivado* Corresponden al tiempo que permanece inhabilitado $\overline{aclr}$ para que el simulador procese una trama de entrada al codificador. Y 80 al tiempo que permanece habilitado $\overline{aclr}$ entre trama y trama..

Constantes a modificar antes de lanzar la simulación.

BER: Cantidad de ruido uniforme en la entrada del decodificador. $0,0 \leq BER \leq 1,0$. Este valor no se expresa en %. Sin embargo debemos tener en cuenta que en los resultados se da BERin, SERin y BERout en %. Se modifica en el subsistema "GeneraRuido".

Los bits de entrada pueden organizarse de manera continua, una sola trama, o en X tramas de Y bits cada una. Para ello hay que modificar 3 constantes en el subsistema "GeneraTramasEntrada GeneraACLR".

1. Asignamos valor a la constante X. Número de tramas de la simulación.

2. ACLR_Desactivado, tenemos que asignarle un valor cumpliendo los requisitos de la *tabla 6.3*. Aunque no es necesario mirar la tabla, en cada uno de los simuladores añadimos una nota junto a la constante, que indica el valor que debemos poner.

3. En el subsistema "Y. Fija el numero de bits de cada trama de entrada", asignamos el valor a Y.

La cadena de bits de entrada y el ruido son pseudoaleatorios, por eso hay que modificar la semilla initial seed en los generadores uniform random number. Hay 3, dos en los puertos de entrada del bloque GeneraRuido y el tercero en el interior del bloque "GeneraTramasEntrada GeneraACLR".

Ficheros de entrada y salida

No hay ninguno, es suficiente con la información de los displays.

6.9 Referencias.

Proporcionamos el enlace Web siempre que exista, accedemos a estos enlaces por última vez en noviembre de 2011. Además tenemos una copia de seguridad en el CD del proyecto de toda la documentación sin copyright.

[1] Documento Xilinx: "ISE In-Depth Tutorial". UG695 (v13.3) October 19, 2011. Páginas 75-130.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_tutorial_ug695.pdf

[2]Documento Xilinx, manual ISE Simulator, ISim: "ISim In-Depth Tutorial". UG682 (v13.3), 19 octubre de 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ug682.pdf

[3]Documento Xilinx: "Synthesis and Simulation Design Guide". UG626 (v 13.3) October 19, 2011. Páginas 7-10; 105-176.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sim.pdf

[4]Francisco Javier López Martínez y Dr. José Tomás Entrambasaguas Muñoz. "Diseño de Transmisor y Receptor para Redes Inalámbricas y W-MAN". Escuela Técnica Superior de Ingeniería de Telecomunicación, Universidad de Málaga. Páginas 3-5. 7 de Julio de 2005.
https://www.coit.es/pub/ficheros/p068_resumen_vodafone_d51dc759.pdf

[5]Todd K.Moon. "Error Correction Coding. Mathematical Methods and Algorithms".Wiley-Interscience, 2005.
[5A] Páginas 4-9; 425-427.
[5B] Páginas 9-34.

[6]Bernard Skalar. "Digital Communications, Fundamentals and Applications". Prentice Hall PTR , segunda edición 21 Enero 2001.
[6A] Páginas 1-11; 461-469.
[6B] Páginas 30-33; 104-136; 167-236.

[7]John G. Proakis. "Digital Communications". McGraw Hill, fourth edition 2001.
[7A] Páginas 1-16; 468-470; 717-724; 830-832.
[7B] Páginas 257-260; 269-274.
[7C] Páginas 17-52; 257-313.

[8]Andrew J.Viterbi and Jim K. Omura. "Principles of Digital Communication and Coding". McGraw-Hill Series in Electrical Engineering, pp 3-35. 1979.

[9]Alain Glavieux. "Channel Coding in Communication Networks. From Theory to Turbocodes". ISTE Ltd, pp1-39. 2007.

[10] D.G. Hoffman; D.A. Leonard; C.C. Linder; K.T. Phelps; C.A. Rodger and J.R. Wall. "Coding Theory. The Essntials". Marcel Dekker, Inc, pp 1-39. 1991.

[11] Stephen G. Wilson. "Digital Modulation and Coding". Prentice Hall Inc, 1996.
[11A] Páginas 3-12; 141-160.
[11B] Páginas 18-42; 76-113; 161-204.
[11C] Páginas 610-611.

[12] Hamid Jafarkhani. "Space-Time Coding. Theory and Practice". Cambridge University Press, pp 16-17. 2005.

[13] Peter Sweeny. "Error control Coding. From Theory to Practice". Wiley 2002, pp 22-23; 56-57; 64; 220-221; 232-235. 2002.

[14] Krishna R. Narayanan and Gordon L. Stüber. "Performance of Trellis-Coded CPM with Iterative Demodulation and Decoding". IEEE Transactions on Communications, Vol 49, Nº4, Abril 2011.
http://www.tamu.edu/faculty/commtheory/papers/trelliscodedCPM_comm01.pdf

[15] Chip Fleming. "A Tutorial on Convolutional Coding with Viterbi Decoding". Spectrum Applications 11 Febrero de 2006.
Enlace Web con las fórmulas de interés:
http://home.netcom.com/~chip.f/viterbi/examples.html
Sitio Web principal: http://pw1.netcom.com/~chip.f/viterbi/tutorial.html
Correo autor: cfleming@ieee.org

[16] Documento Xilinx: ""System Generator for DSP. Reference Guide". UG638 (v 13.3) October 19, 2011.
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sysgen_ref.pdf

[16A] Páginas 413-417. *Nota 6.8*: Se refiere a la versión Viterbi decoder 7.0 y en nuestro código usamos la 5.0. Nos vemos obligados a utilizar la 5.0 porque es la versión disponible en el software del que disponemos: System Generator v8.1. Sin embargo, en el momento de redactar este documento, diciembre 2011, no estaba disponible en la Web la documentación del 5.0, sólo estaba accesible vía Web la documentación del 7.0. Pero no supone ningún problema, porque la documentación del 5.0 siempre estará disponible en la ayuda de System Generator. No la referenciamos porque la ayuda no es accesible vía Web. Pero sí la incluimos en la documentación que entregamos al realizar el CD del proyecto.

[16B] Páginas 419-429.

[17] Michael Francis. "Viterbi Decoder Block Decoding – Trellis Termination and Tail Biting". Documentación de Xilinx, XAPP551 (v2.0), 30 de Julio de 2010.
http://www.xilinx.com/support/documentation/application_notes/xapp551.pdf

[18] Christian B. Schlegel and Lance C. Pérez. "Trellis and Turbo Coding". IEEE Press Series on Digital & Mobile Communication, pp. 138-141, 2004.

CAPÍTULO 7

7. COMPARATIVA DECODIFICADORES. RESULTADOS SÍNTESIS Y SIMULACIÓN.

7.1 Objetivos.

En los capítulos anteriores hemos descrito los dos decodificadores que hemos realizado: UC3M ,(*ViterbiDecoder.vhd*) y Opencores, (*decoderverilog.v*).

En el *capítulo 6* describimos el simulador que hemos desarrollado y con el que comprobamos si los Viterbis funcionan correctamente.

En este capítulo compararemos los dos decodificadores entre sí y con el que utilizamos como referencia, IP core Viterbi decoder v5.0 de Xilinx. La documentación de Xilinx está disponible en [1], [2], [3] y [4].

En los *apartados 7.2 a 7.6* comparamos las características relativas al hardware: área ocupada, frecuencia máxima de trabajo. Prestamos atención a estos puntos:

- Explicamos los detalles de la arquitectura, la razón de usar esa arquitectura y las alternativas que existen.
- Los pasos que hemos dado para optimizar el compromiso entre área y velocidad.
- Los parámetros del decodificador que pueden modificarse, como realizar las modificaciones, y la relación entre coste y beneficio de esa modificación

En el *apartado 7.7* comparamos la funcionalidad de los 3 decodificadores. Para ello, gracias a los simuladores que hemos realizado, medimos la BER a la salida con diversos niveles de relación Eb/No en la entrada.

Para realizar la comparación de rendimiento formamos gráficas que relacionan BER con Eb/No. Referencias sobre las gráficas en [5], [6], [7], [34], [35], [36], [37], [38] y [39]. En estas gráficas también se puede ver como varía la capacidad de corrección de errores dependiendo de estas variables.:

1. Decoding depth.
2. Número de bits de cada trama.
3. Presencia de errores de ráfaga.

También explicamos los pasos a seguir para poder asegurar que el *ViterbiDecoder.vhd* implementa un algoritmo Viterbi exacto. Hemos superado con éxito todos esos pasos, por lo que hemos cumplido con los objetivos del proyecto.

7.2 Síntesis UC3M, ViterbiDecoder.vhd.

Una característica fundamental es que todo el código está realizado íntegramente en VHDL sin utilizar elementos dependientes de la tecnología. No instancia ninguna primitiva ni referencia a arquitecturas o recursos específicos de un dispositivo hardware. Lo hemos realizado así para cumplir un requisito del proyecto, que el diseño sea completamente multiplataforma.

En el laboratorio disponemos de una tarjeta Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 xc4vsx55:

Speed grade = -10 y encapsulado ff1148.
Datasheets y guías de usuario en [8], [9],[10], [11] y [12].

Por este motivo todo el proceso de síntesis e implementación lo realizamos siempre sobre esta tarjeta. Todos los resultados que mostramos en esta memoria se refieren a esta tarjeta en concreto.

7.2.1 Resumen resultados tras síntesis y map.

Nuestra especificación es la de un decodificador Viterbi con decoding depth =36. Sólo se muestran los del bloque principal, los resultados de síntesis del resto de bloques están en el *anexo D*.

Tabla 7.1: Síntesis ViterbiDecoder.vhd. (Decoding depth = 36)			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	7060	24576	28%
Number of Slice Flip Flops	3712	49152	7%
Number of 4 input LUTs	12488	49152	25%
Number of bonded IOBs	8	640	1%
Number of FIFO16/RAMB16s	32	320	10%
Number used as RAMB16s	32		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period, T_{CLK}/ Maximum Frequency, F_{CLK}	9,912 ns		100,884 MHz
Velocidad máxima de decodificación. *Nota 7.1*	12,615 Mega símbolos/s		
Minimum input arrival time before clock	8,852 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Nota 7.1: El decodificador puede trabajar con una frecuencia máxima de 100,884 MHz. Pero hay que tener en cuenta que emplea 8 T_{CLKs} en decodificar un símbolo. Por tanto la velocidad de decodificación es 8 veces menor.

En la entrada hay símbolos de 2 bits, mientras que en la salida cada símbolo de 2 bits equivale a un dato de un bit. Entonces la velocidad máxima es de 12,615 mega símbolos, que equivale a 12615 Kbits/s en la salida y 2*12615 Kbits/s en la entrada.

Tabla 7.2: Map ViterbiDecoder.vhd. (Decoding depth = 36)			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	3707	49152	7%
Number of 4 input LUTs	12047	49152	24%
Number of occupied slices	8383	24576	34%
Number of Slices containing only related logic	8383	8383	100%
Number of Slices containing unrelated logic	0	8383	0%
Total Number 4 input LUTs	12098	49152	24%
Number used as logic	12047		
Number used as rote-thru	51		
Number of bonded IOBs	8	640	1%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Number of FIFO16/RAMB16s	32	320	10%
Number used as FIFO16s	0		
Number used as RAMB16s	32		
Total equivalent gate count for design	122714		

Tras realizar la síntesis, map y place&route de nuestro diseño, llegamos a las siguientes conclusiones:

1. No hay errores en la síntesis ni en la implementación.
2. Hemos cumplido todas las reglas del diseño síncrono. [13], [14].
3. No hay ningún latch. [15], [16].
4. Aparecen warnings, pero los analizamos todos y nos aseguramos de que todos están permitidos. Ninguno de ellos afecta a la arquitectura ni al funcionamiento.

Los 4 puntos anteriores nos indican que hemos cumplido correctamente el flujo de diseño del *apartado 1.8*. Referencias [17], [18] y [19] al final de este capítulo.

7.2.2 Elección de arquitectura combinada, 8 etapas serie*8 módulos paralelo.

Debemos elegir el mejor compromiso entre área y velocidad. La estructura serie es la más lenta pero la que menos área ocupa. Por el contrario la paralelo es la más rápida, pero la que más área ocupa. Referencias [20], [21], [22] y páginas 18, 19, 29 y 30 de [3]

Hemos elegido una arquitectura combinada, en la que se emplean (8 etapas serie)*(8 módulos paralelo). Porque es con la que hemos obtenido un mejor compromiso entre área y velocidad.

Una ventaja del diseño es que emplea una única frecuencia de reloj, $F_{CLK}=1/T_{CLK}$.

El decodificador debe analizar 64 estados para decodificar cada uno de los símbolos de entrada. Este análisis no puede solaparse, de manera que hasta que no se han terminado de procesar los 64 estados correspondientes a un símbolo de entrada, no se puede comenzar el análisis del siguiente símbolo.

Para realizar este trabajo el decodificador está dividido en 8 etapas serie y en cada una se procesan en paralelo 8 estados. Cada etapa serie tiene un tiempo de proceso de 1 T_{CLK}, de manera que se necesitan 8 T_{CLKs} para procesar un símbolo. A este tiempo lo denominamos período de proceso de un dato.

Entonces el decodificador trabaja con un una frecuencia $F_{CLK} = 1/T_{CLK}$. Pero cada símbolo de entrada debe llegar con una frecuencia 8 veces menor. Así que en la salida se obtiene un nuevo bit decodificado cada 8 T_{CLKs}. Por este motivo la velocidad de decodificación es igual a $F_{CLK}/8$.

En nuestro decodificador el bloque ACS está constituido por módulos más pequeños, DistanceJIandHI y DistanceLastStateBitIn. En cada uno de estos módulos se procesa un estado en un T_{CLK}. Entonces es fácil construir estructuras diferentes, ejemplo una estructura todo serie, todo paralelo... Hemos utilizado esta ventaja para probar con otro tipo de estructuras hasta que finalmente comprobamos que la combinación más eficiente es la que usamos de 8*8.

La base está en que en cada módulo serie se procesan un número de estados igual al número de bloques, DistanceJIandHI y DistanceLastStateBitIn, que se hayan instanciado en esa etapa serie. Cada etapa serie emplea 1 T_{CLK}.

Entonces, por ejemplo para construir un decodificador serie emplearíamos sólo un módulo DistanceJIandHI y un DistanceLastStateBitIn, y en cada T_{CLK} procesaríamos sólo un estado. El consumo de área es mínimo, pero la velocidad de decodificación también, puesto que sería de $F_{CLK}/64$.

Para construir uno paralelo emplearíamos 64 módulos DistanceJIandHI y 64 DistanceLastStateBitIn. De manera que en un sólo T_{CLK} se procesan los 64 estados correspondientes a un dato, pero a cambio empleamos más área.

Un último ejemplo, para tener un decodificador de 4 etapas serie y 16 paralelo, instanciaríamos 16 módulos distanceJIandHI y 16 DistanceLastStateBitIn. En cada bloque serie se procesarían 16 estados, y la velocidad de decodificación sería de $F_{CLK}/4$.

7.2.3 Optimización frecuencia máxima.

Hemos optimizado el código para obtener la máxima frecuencia. Finalmente hemos conseguido F_{CLK} = 100,884 MHz. Con la que obtenemos una velocidad máxima de decodificación de 12615 Kbits en la salida.

Todos los bloques están optimizados, pero especialmente el que constituye el camino crítico, puesto que es el que limita la frecuencia. El camino crítico está formado por el bloque FindMinIndex. Por eso en su interior realizamos los cálculos en tres etapas registradas utilizando la técnica pipeline, [46], [47] y [48], gracias a la cual conseguimos aumentar F_{CLK}.

7.2.4 Optimización área.

La disminución del área ha sido lo que más esfuerzo nos ha costado. El decodificador debe almacenar 64 vectores con decoding depth bits cada uno. Y son vectores dinámicos, en un período de proceso hay que leerlos todos y volverlos a escribir. De manera que entre un período de proceso y el siguiente hay que modificar los 64 vectores.

Hemos optimizado el área en todos los módulos del decodificador, pero en el que más tiempo hemos invertido ha sido en el CalculatePaths.

CalculatePaths se encarga de almacenar y gestionar todas las lecturas y escrituras en los 64 vectores. Inicialmente lo desarrollamos empleando registros para almacenar los bits contenidos en los vectores. Este decodificador era más intuitivo a la hora de implementar el código, pero no era viable técnicamente porque ocupaba mucho área.

Por este motivo desechamos ese CalculatePaths inicial realizado con registros, y diseñamos otro que emplea la memoria RAM de la FPGA para almacenar los vectores. Es el que empleamos en nuestro decodificador final, *ViterbiDecoder.vhd*, en el que el área está optimizada. Se utilizan mejor los recursos hardware de la FPGA, como se ve en la siguiente tabla:

Tabla 7.3: Comparativa Síntesis CalculatePaths.vhd					
Design Summary. (Nota 7.2: Decoding depth = 36)					
	Hecho con registros			Hecho con RAM	
Logic Utilization	Used	Available	Utilization	Used	Utilization
Number of Slices	7066	24576	28%	2454	9%
Number of Slice Flip Flops	4788	49152	9%	890	1%
Number of 4 input LUTs	13570	49152	27%	4765	9%
Number of bonded IOBs	261	640	40%	261	40%
Number of FIFO16/RAMB16s	0	320	0	32	10%
Number used as RAMB16s	0			32	
Number of GCLKs	1	32	3%	1	3%
Timing Summary					
Minimum Period, T_{CLK}/Maximum F_{CLK}	3,798ns	263,307 MHz		5,898ns	169,61 MHz
Minimum input arrival time before clock	9,360 ns			9,864 ns	
Maximum output required time after clock	4,869 ns			4,851 ns	
Maximum combinational path delay	No Path found			No path found	

El inconveniente es que para optimizar el área hemos tenido que realizar un código muy complejo y nada intuitivo, por lo que nos llevó mucho tiempo su diseño. Pero el esfuerzo está justificado a la vista de los resultados de la tabla anterior.

El decodificador hecho con registros no sería más rápido que el realizado con RAM. Porque aunque el CalculatePaths tenga mayor F_{CLK}, esto no influye, ya que no está en el camino crítico.

En el *apartado 5.5.11*, explicamos que en el *ViterbiDecoder.vhd* empleamos dos memorias RAM con 64*(decoding depth) bits cada una. Y a su vez, cada memoria RAM se divide en bloques más pequeños de 4*(decoding depth) bits. Entonces en el *ViterbiDecoder.vhd* se emplean 32 bloques de memoria RAM, con 4*(decoding depth) bits cada uno. Por eso tras realizar la síntesis y el map en los resultados aparece una utilización de 32 módulos de FIFO16/RAMB16s.

Otro dato importante es que para conseguir que el diseño sea multiplataforma no generamos los módulos de RAM mediante un core, que sería lo más sencillo. Sino que nos vemos obligados a generarlo mediante sentencias VHDL.

Justificación de que la organización de la RAM en bloques de 4 filas es la más óptima:

En el caso de la Virtex 4, hay disponibles 320 módulos, que pueden organizarse en: 4k*4 bits, 2k*9 bits, 1k*18 bits y 512*36 bits. Consultar datasheet [12], página 115.

Como en el código se emplean bloques de 4*36 bits, al sintetizar, cada uno de esos bloques se implementa físicamente en un módulo de 512*36 bits. Por eso en este diseño se utilizan 32 FIFO16/RAMB16s. Esto no es eficiente en términos de área porque en cada módulo de RAM sólo se utilizan 4 filas, dejando 508 libres. Pero está justificado para conseguir una F_{CLK} adecuada, como explicamos a continuación.

Para utilizar los recursos de memoria de una manera más eficiente habría que utilizar bloques de memoria con más filas. La memoria podría organizarse en el código en sólo 2 bloques de 64*36 bits. De esta manera, al sintetizar, sólo se emplearían 2 módulos de RAM de la tarjeta Virtex 4. Sin embargo esto no sería viable, porque al hacerlo disminuiría mucho la frecuencia máxima del decodificador, como indicamos a continuación:

- En el diseño definitivo, codificamos bloques de 4*36 bits, y la frecuencia máxima de CalculatePaths es de 169,61 MHz. Entonces CalculatePaths no está en el camino crítico, la frecuencia máxima es de 100,884 MHz y la fija FindMinIndex.

- Codificar bloques de 8*36 bits disminuye a 16 el número de módulos necesarios en la tarjeta. Pero a cambio la frecuencia máxima de CalculatePaths sería aproximadamente la mitad, quedándose en cerca de 85 MHz. En este caso el camino crítico lo marcaría CalculatePaths, y la frecuencia máxima del decodificador sería por tanto de unos 85 MHz.

- Si codificásemos bloques de 16*36 bits el numero de módulos necesarios sería de 8, pero la frecuencia máxima del decodificador se vuelve a dividir por un orden de 2, quedando en torno a los 42 MHz.

- Empleando bloques de 32*36 bits se necesitarían 4 módulos de RAM, pero la frecuencia máxima del decodificador sería del orden de 21 MHz. Y con bloques de 64*36 bits necesitamos 2 módulos y la frecuencia sería 10,5 MHz.

Por tanto los resultados anteriores nos muestran que la mejor opción es la que hemos elegido, bloques de 4*36 bits.

7.2.5 Modificación decoding depth.

Como valor añadido al proyecto, además de un decodificador que cumpla las especificaciones, con decoding depth = 36, hemos implementado otros con decoding depth = 8, 24, 32, 48 y 60.

Una gran ventaja del código es que modificar el decoding depth es muy simple, puesto que sólo hay que modificar las constantes DecodingDepth y DecodingDepth_1, situadas en *constantes.vhd*. Con esto tenemos el *ViterbiDecoder.vhd* configurado. Sin embargo si queremos simularlo, hay que cambiar otra constante en el simulador, porque al cambiar el decoding depth se modifica la latencia del decodificador. En el manual de usuario, *apartados 6.6.2 y 6.8.2* se indican los pasos a seguir.

Tabla 7.4: Síntesis ViterbiDecoder.vhd. Decoding depth 8, 24, 32, 36, 48, 60						
Design Summary						
Logic Utilization						
Decoding depth	8	24	32	36	48	60
Number of Slices	5914	6461	6575	7060	7577	8323
Number of Slice Flip Flops	3250	3510	3645	3712	3909	4102
Number of 4 input LUTs	9741	11247	12048	12488	1631	14763
Number of bonded IOBs	8	8	8	8	8	8
Number of FIFO16/RAMB16s	32	32	32	32	64	64
Number used as RAMB16s	32	32	32	32	64	64
Number of GCLKs	1	1	1	1	1	1
Timing Summary						
Minimum period, T_{CLK}/ Maximum Frequency (MHz), F_{CLK}	9,912 ns / 100,884 MHz					
Velocidad de decodificación	12,615 Mega símbolos/s					
Minimum input arrival time before clock	8,852 ns					
Maximum output required time after clock	4,851 ns					
Maximum combinational path delay	No path found					

Tras analizar la tabla anterior llegamos a las siguientes conclusiones:

- La frecuencia máxima de trabajo no cambia. Porque está limitada únicamente por el bloque FindMinIndex y este bloque es independiente del decoding depth.

- Hemos conseguido que el área del decodificador, slices y LUTs, se mantenga sin apenas variaciones a pesar de que hagamos grandes cambios en el decoding depth. Esto es una ventaja más de implementar los vectores que contienen los caminos con RAM en vez de con registros. En caso de haberlos hecho con registros, el número de slices y de LUTs crecería muchísimo al aumentar decoding depth.

- El número de módulos de RAMB16s es el doble si decoding depth es mayor que 36. Esto se debe a que el tamaño máximo de palabra de un módulo de RAM de la Virtex 4 es de 36 bits, consultar página 115 de [12]. Por eso cada bloque de 4*(decoding depth) bits necesita dos módulos físicos de memoria en la FPGA.

7.2.6 Modificación parámetros decodificador.

- El único parámetro que se puede modificar es el decoding depth. Es inmediato, sólo hay que asignar valor a una constante.

- Constraint length K=7, el número de estados y decode rate R = 1/2 no se pueden modificar, porque son la base de la arquitectura. Un cambio en estos parámetros exigiría grandes cambios en el código.

- El polinomio generador tampoco. Aunque no tiene importancia porque ya hemos elegido el más óptimo, el que maximiza la distancia mínima de Hamming (d_{min} ó d_{free}). No tiene sentido hacer un decodificador Viterbi de 64 estados y R = 1/2 con otro polinomio. Referencias [27] y [28].

- El período de proceso es de 8 T_{CLKs}. Lo hemos elegido así para obtener el mejor compromiso entre área y velocidad. Se puede modificar pero habría que hacer cambios importantes en la arquitectura.

7.3 Síntesis UC3M, SimulacionCompleta.vhd

Se trata del simulador que utilizamos, que incluye al decodificador, el codificador y todos los elementos auxiliares. En el diseño definitivo en la FPGA sólo se sintetizará el decodificador. Sin embargo durante la fase de pruebas se puede sintetizar el simulador en la placa, para realizar las pruebas no sólo con el PC, sino también sobre el hardware real. Este proceso se denomina emulación.

Hemos seguido la misma metodología de diseño que con el *ViterbiDecoder.vhd* y tras la síntesis comprobamos que:

1. No hemos usado elementos dependientes de la tecnología, el simulador es totalmente multiplataforma.
2. No hay errores en la síntesis ni en la implementación.
3. Hemos cumplido todas las reglas del diseño síncrono. [13], [14].
4. No hay ningún latch. [15], [16].
5. No hay nuevos warnings añadidos respecto a los del *ViterbiDecoder.vhd*.

Tabla 7.5: Síntesis SimulacionCompleta.vhd. (Decoding depth = 36)				
Design Summary	SimulacionCompleta			ViterbiDecoder
Logic Utilization	Used	Available	Utilization	
Number of Slices	7223	24576	29%	7060
Number of Slice Flip Flops	3900	49152	7%	3712
Number of 4 input LUTs	13233	49152	25%	12488
Number of bonded IOBs	16	640	2%	8
Number of FIFO16/RAMB16s	32	320	10%	32
Number used as RAMB16s	32			32
Number of GCLKs	1	32	3%	1
Timing Summary				
Minimum period, T_{CLK} / Maximum F_{CLK}	10,204 ns	97,997 MHz		100,884 MHz
Velocidad máxima de simulación.	12,25 Mega bits/s			12,615 Mega símbolos/s
Minimum input arrival time before clock	9,6751 ns			8,852 ns
Maximum output required time after clock	5,845 ns			4,851 ns
Maximum combinational path delay	6,399 ns			No path found

Tabla 7.6: Map SimulacionCompleta.vhd. (Decoding depth = 36)				
Design Summary	SimulacionCompleta			ViterbiDecoder
Logic Utilization	Used	Available	Utilization	
Number of Slice Flip Flops	3897	49152	7%	3707
Number of 4 input LUTs	12733	49152	25%	12047
Number of occupied slices	8933	24576	36%	8383
Number of Slices containing only related Logic	8933	8933	100%	8383
Number of Slices containing unrelated logic	0	8933	0%	0
Total Number 4 input LUTs	12817	49152	26%	12098
Number used as logic	12733			12047
Number used as rote-thru	84			51
Number of bonded IOBs	16	640	2%	8
Number of BUFG/BUFGCTRLs	1	32	3%	1
Number used as BUFGs	1			1
Number used as BUFGCTRLs	0			0
Number of FIFO16/RAMB16s	32	320	10%	32
Number used as FIFO16s	0			0
Number used as RAMB16s	32			32
Total equivalent gate count for design	129374			122714

Comparando *SimulacionCompleta* con *ViterbiDecoder* vemos que la frecuencia máxima apenas ha disminuido, y que el aumento de área es pequeño, del orden del 5 %. Esto se debe a que también hemos optimizado el simulador en área y en velocidad.

7.4 Síntesis Opencores, decoderverilog.v.

Este decodificador no es válido porque no implementa el algoritmo de Viterbi exacto, y además tiene latches. En el *capítulo 4* explicamos los errores de este decodificador y la razón por la que lo descartamos.

La documentación de este decodificador está disponible en la referencia [29].

A continuación mostramos los resultados tras la síntesis:

Tabla 7.7: Síntesis decoderverilog.v			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	1671	24576	6%
Number of Slice Flip Flops	909	49152	1%
Number of 4 input LUTs	2275	49152	4%
Number used as logic	1251		
Number used as RAMs	1024		
Number of bonded IOBs	8	640	1%
Number of FIFO16/RAMB16s	0	320	0%
Number used as RAMB16s	0		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period, T_{CLK}/ Maximum Frequency, F_{CLK}	8,743 ns		114,378 MHz
Velocidad de decodificación.	14,297 Mega símbolos/s		
Minimum input arrival time before clock	6.850 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

- El período de proceso es de 8 T_{CLKs}, velocidad de decodificación es $F_{CLK}/8$.
- Ocupa menos área y el ligeramente más rápido que el *ViterbiDecoder.vhd*. Pero esta ventaja no justifica el uso de este decodificador.

7.5 Comparativa de latencias en los 3 decodificadores.

Tabla 7.8: Latencias de todos los diseños.										
Decodificador	Decoding depth									
	24		32		36		48		60	
	S	D	S	D	S	D	S	D	S	D
UC3M	26,5	26,25	34,5	34,25	38,5	38,25	50,5	50,25	62,5	62,25
Opencores			36,875	36,625						
Xilinx (paralelo)	115	114			163	162	211	210	259	258

- La latencia está medida en períodos de proceso de un símbolo. Equivale a 8 T_{CLKs} en UC3M y Opencores y a $1T_{CLK}$ en Xilinx.

- La columna D corresponde a la latencia entre EnableInDecoder y EnableOutDecoder. Y la S entre EnableInEncoder y EnableOutDecoder.

- Medimos las latencias de todos los decodificadores usando nuestros simuladores, ver *figura 7.1*. Pero además la latencia del Xilinx puede consultarse en la página 28 de su datasheet [3].

En el diseño UC3M tenemos estas latencias en los bloques de interés:

- ViterbiDecoder = [10 + (decoding depth+1)*8] T_{CLKs}
- EncoderK7 = 1 T_{CLK}.
- MantienePulso = 1 T_{CLK}.

El propio algoritmo Viterbi exige que la latencia mínima ideal es igual al decoding depth. Porque no se pueden empezar a sacar bits decodificados del trellis hasta que éste no se haya rellenado con decoding depth símbolos. Así que nuestro decodificador tiene una latencia muy buena, sólo 2,25 símbolos adicionales frente al valor mínimo ideal.

En cambio en este apartado el decodificador de Xilinx es claramente inferior, obteniendo latencias del orden de cuatro veces mayores que en el UC3M.

Si se mide la latencia en T_{CLKs} entonces el de Xilinx tiene menos que el UC3M. Por ejemplo con 36 posiciones de memoria en el UC3M se obtiene una latencia de 306 T_{CLKs}, 38,25 períodos de proceso. En cambio en el de Xilinx se obtienen 162 T_{CLKs} y 162 períodos de proceso.

Pero en esta característica en concreto el valor en T_{CLKs} no importa, lo que interesa es cuántos símbolos tienen que llegar a la entrada del decodificador antes de obtener el primer bit decodificado en la salida. Y esta medida se corresponde a la latencia en períodos de proceso. Por tanto podemos concluir que el UC3M tiene un valor muy bueno de latencia, del orden de 4 veces menor que el de Xilinx.

La clave está en que si a la entrada de los decodificadores llegan símbolos con período T_{CLK2}, el decodificador de Xilinx trabajará con T_{CLK2}, mientras que el de Xilinx lo hará

con $T_{CLK1} = T_{CLK2}/8$. Entonces en UC3M tendremos una latencia $306T_{CLK1}=38{,}25T_{CLK2}$. Y en el de Xilinx tendremos 162 T_{CLK2}.

En la siguiente figura se aprecia cómo cuando a los dos decodificadores llegan los mismos datos y al mismo tiempo, el UC3M obtiene los primeros resultados mucho antes que el Xilinx.

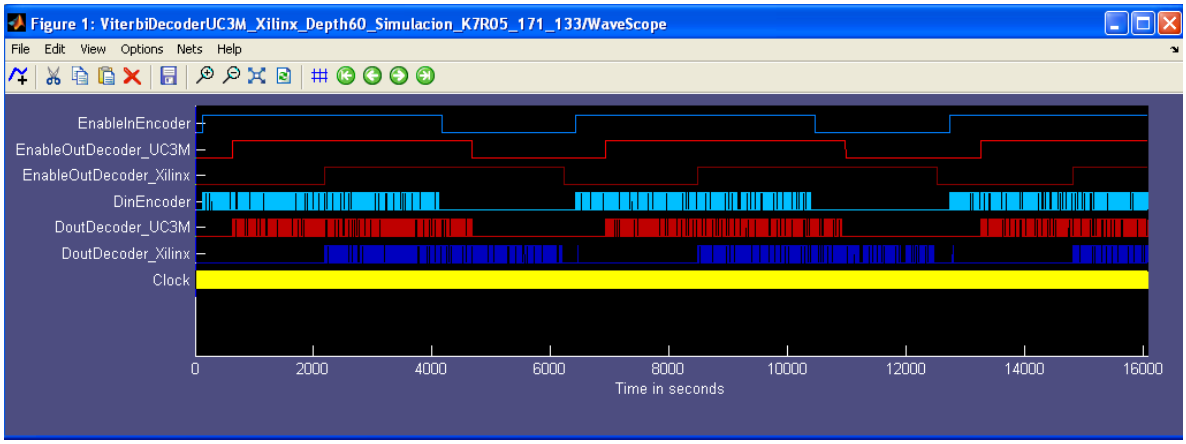


Figura 7.1: Comparativa latencias UC3M y Xilinx.

7.6 Síntesis Xilinx IP core Viterbi decoder 7.0.

Para utilizar el Viterbi de Xilinx en un dispositivo hardware es necesario comprar una licencia, full license key. (Consultar páginas 13 y 14 de [4]). No disponemos de ella, porque el objetivo principal del proyecto es desarrollar nuestro propio decodificador.

Pero se puede adquirir una licencia de evaluación gratuita, full system hardware evaluation license key , válida durante 120 días. Esta licencia permite realizar todo el proceso de síntesis y simulación. También permite descargar el código en la tarjeta, pero el código sólo funcionará en la tarjeta durante un tiempo limitado (2 horas).

Nosotros hacemos uso de esta licencia para sintetizar el decodificador de Xilinx y así poder compararlo con nuestros resultados. El proceso de obtención de licencias puede consultarse en la referencia [2], en las páginas 13 y 14 de [4] y en [30], [31] y [32].

Nota 7.3: Los resultados de síntesis corresponden a la versión 7.0. Pero en la simulación hemos usado la 5.0. Esto se debe a que simulamos con la versión 8.1 de System Generator, en la cual sólo se dispone del Viterbi decoder 5.0. Sin embargo en el software de Xilinx ISE sólo disponemos de la licencia del Viterbi decoder 7.0.

Nota 7.4: En las páginas 29 y 30 del datasheet del Viterbi de Xilinx, [3], se muestran los resultados de síntesis del decodificador. Pero sólo nos valen para hacernos una idea aproximada, porque no se obtienen sintetizando sobre la FPGA Virtex 4 XC4VSX55, que usamos en el laboratorio. Documentación de la tarjeta en [8], [9],[10], [11] y [12]. Por este motivo hemos adquirido la licencia de evaluación gratuita, de esta manera podemos sintetizar el decodificador de Xilinx sobre la Virtex 4 XC4VSX55, y obtener así los resultados exactos.

Tabla 7.9: Síntesis Viterbi decoder v7.0 de Xilinx.						
Logic Utilization						
Común: K=7, R=1/2, 64 estados. Polinomio 171, 133 octal	Paralelo				Serie	
	Decodificación dura		Decodificación soft 3 bits		Decodificación soft 3 bits	
Decoding depth	36	60	36	60	36	60
Number of Slices	2074	2082	2750	2775	2010	2031
Number of Slice Flip Flops	1813	1822	2537	2561	2298	2318
Number of 4 input LUTs	3700	3717	4953	4994	3498	3536
Number used as logic	3660	3665	4880	4898	3176	3190
Number used as shift registers	24	36	49	72	318	342
Number used as RAMs	16	16	24	24	4	4
Number of IOs	44		56		48	
Number of bonded IOBs	44		56		48	
Number of FIFO16/RAMB16s	5		8		2	
Number used as RAMB16s	5		8		2	
Number of GCLKs	1		1		1	
Timing Summary						
Minimum period (ns) T_{CLK}	5,785		5,756		5,869	
Maximum Frequency (MHz) F_{CLK}.	172,862		173,739		170,388	
Velocidad de decodificación. Mega símbolos/s	172,862		173,739		14,199	
Minimum input arrival time before clock (ns)	9,493	9,497	9,907	9,92	9,497	9,516
Maximum output required time after clock (ns)	6,854		6,854		5,421	5,141
Maximum combinational path delay	No path found					

En la arquitectura paralelo el período de proceso de un símbolo es de un T_{CLK}. Por eso la velocidad de decodificación coincide con F_{clk}. En cambio en la serie es de 12 T_{CLKs}, así que la velocidad de decodificación es 12 veces menor, (ver páginas 29 y 30 de [3]).

En el UC3M la velocidad de decodificación es de 12,615, equivalente por tanto a la del dispositivo serie. Pero ocupa más área que cualquiera de los modelos de Xilinx. Hemos optimizado nuestro decodificador en área y velocidad, pero el rendimiento es peor que en el modelo de Xilinx. Hay dos razones que lo justifican:

1. El de Xilinx sólo puede utilizarse en determinadas FPGAS de la familia Xilinx, así que está optimizado para trabajar en un hardware concreto y sobre él podrá trabajar a más velocidad y ocupando menos área. En cambio nosotros hemos optado por hacerlo multiplataforma, puede trabajar en cualquier dispositivo hardware: ASICs, FPGAs, CPLDs, PLDs... ([23], [24], [25] y [26] para ampliar información sobre los tipos de dispositivos). Entonces el UC3M sobre una FPGA Xilinx no será tan rápido como el Viterbi de Xilinx, porque no está optimizado para esta arquitectura específica. Pero contamos con la ventaja de que nuestro decodificador se puede implementar en cualquier otro dispositivo.

2. Hemos optimizado el área y la velocidad cumpliendo sobradamente las especificaciones requeridas en el proyecto. Alcanzar las prestaciones de Xilinx sobrepasa ampliamente la ambición de este proyecto.

3. Disponemos de otra opción si queremos aumentar la velocidad y disminuir el área. Consiste en emplear un hardware más eficiente, como un ASIC. Con un ASIC tecnológicamente equivalente a la Virtex 4, (CMOS cobre 90 nm), la velocidad se multiplicaría aproximadamente por 3,1 el área total se dividiría por 37 y el consumo se dividiría por 9,2. Referencia [33].

7.7 Gráficas BER frente a Eb/No.

7.7.1 Verificación de que el UC3M es funcionalmente correcto.

En el *apartado 7.2* hemos demostrado que el código que hemos desarrollado puede sintetizarse en hardware sin ningún problema. Ahora falta demostrar que el código realiza correctamente la función para la que ha sido diseñado. Para ello disponemos de los simuladores que hemos implementado y sometemos a nuestro decodificador a diversas pruebas.

Primera prueba, verificación de que el código convolucional es reversible.

En la *figura 6.1*, la secuencia original se codifica convolucionalmente, obteniéndose c[n] = F(m[n]). Esta secuencia se transmite y en el receptor se recibe r[n] que es la entrada al decodificador. En ausencia total de ruido, la secuencia recibida debe ser exactamente igual a la transmitida, r[n] = c[n- $retardo_{canal}$].

Cuando se den estas condiciones, la secuencia a la salida del decodificador debe ser exactamente igual a la original de entrada al codificador retardada:

$$DoutViterbi[n] = F^{-1}(r[n - retardo_{decoder}]) = F^{-1}(c[n - retardo_{decoder+canal}]) \Rightarrow$$
$$DoutViterbi[n] = F^{-1}(F(m[n - retardo_{decoder+canal}])) = m[n - retardo_{decoder+canal}]$$

La secuencia m[n] puede ser continua o dividida en tramas, en todos los casos debe cumplirse la condición de igualdad de la ecuación anterior.

Para hacer esta prueba utilizamos los simuladores de Xilinx y de System Generator. En este caso no es necesario utilizar los simuladores en los que están instanciados al mismo tiempo el decodificador UC3M y el de Xilinx, es suficiente con emplear cualquiera de los que instancian únicamente el UC3M. La prueba consiste en utilizar BERinDecoder=0,0, y lanzar una simulación de varios millones de bits.

El resultado de la simulación debe ser obligatoriamente de cero errores. Un error indica que hay un bit diferente entre la entrada al codificador y la salida del decodificador. En este caso las dos cadenas deben ser exactamente iguales, por lo que si el resultado de la simulación no fuese cero errores, no se cumpliría la condición de código reversible.

Esta prueba la realizamos de manera exhaustiva para cada uno de los decodificadores que hemos implementado, con decoding depth 24, 32, 36, 48, 60 y Opencores. Además para cada uno de ellos realizamos tres simulaciones: trama continua, trama de 2000 y de 200 bits. Simulamos en cada caso varios millones de bits y en todos ellos hemos obtenido como resultado cero errores. Con esto podemos concluir que se cumple la primera condición que buscábamos.

Segunda prueba, el rendimiento debe ser igual al decodificador de Xilinx.

El decodificador UC3M debe contener el algoritmo Viterbi exacto. Esto significa que su rendimiento debe ser igual al de los decodificadores comerciales que implementan el mismo algoritmo. Entonces lo comparamos con nuestra referencia, el de Xilinx y debemos obtener los mismos resultados.

Al realizar la comparación hay que tener en cuenta que la decodificación Viterbi tiene cierto carácter aleatorio. Esto significa que ante una misma cadena de entrada, dos Viterbis pueden obtener una cadena de salida diferente. De manera que si la entrada a ambos es r[n], se cumple:

$$\left.\begin{array}{l} \text{DoutViterbi}_1[n] = F^{-1}(r[n]) = x_1[n] \\ \text{DoutViterbi}_2[n] = F^{-1}(r[n]) = x_2[n] \end{array}\right\} \text{El algoritmo } F^{-1} \text{ es igual, pero puede ocurrir } x_1[n] \neq x_2[n]$$

En los dos casos se aplica la decodificación Viterbi correctamente, pero se obtienen cadenas de bits diferentes en la salida. Esto se debe a que el algoritmo se basa en formar caminos en la malla trellis y elegir como camino superviviente ganador el que tenga la menor distancia. Pero si la distancia mínima la comparten varios caminos supervivientes, se elige aleatoriamente uno de ellos.

Además a la hora de formar los caminos se elige la rama ji ó hi que tenga una menor distancia. Pero en el caso de que las dos ramas tengan la misma distancia, se elige aleatoriamente una de ellas.

No hay ninguna norma que indique que rama o camino elegir cuando se dan estas situaciones de igualdad, se puede elegir cualquiera indistintamente. Por tanto, pueden darse situaciones en las que un decodificador elegirá una rama o camino y el otro elegirá ramas o caminos diferentes. Los dos han realizado correctamente el algoritmo, sin embargo el resultado no tiene porque coincidir. Como la decisión es aleatoria, en unos casos se acierta y el decodificador corrige los errores en los bits decodificados. Pero en otros casos no se acierta en la decisión, y habrá bits erróneos en la salida.

Esta regla tiene dos matices:

1. Si la cadena r[n] coincide exactamente con la codificada c[n], en ese caso nunca se dará una situación de ambiguedad en la decodificación. Las cadenas de salida de ambos decodificadores deben ser iguales entre sí y también iguales a la original de entrada al codificador. En este caso el camino superviviente ganador tiene distancia cero y es único, así que no hay posibilidad de error en su elección. Además tampoco hay dudas al formar el camino con distancia cero, porque sólo una de las dos ramas que lleguen hasta él, ji ó hi tendrá distancia cero.

2. En el caso de que en el canal haya ruido tendremos r[n] ≠ c[n]. En este caso es cuando pueden darse las situaciones de ambiguedad. Pero debe cumplirse que cuando se hayan alcanzado un cierto número de errores, las BERouts de los dos decodificadores converjan hacia valores similares. No puede ocurrir que uno de ellos tome siempre la decisión aleatoria acertada que permite decodificar el bit

correctamente y el otro tome siempre la decisión desacertada. Esta teoría dice que la suerte entre ambos debe repartirse, por lo que el número final de errores no será igual, pero sí parecido.

Además hay un aspecto muy importante, consiste en que hemos diseñado el decodificador UC3M con la técnica truncamiento de trellis mientras que el de Xilinx emplea cola de ceros, [40] y [41]. Debido a esto el comportamiento de ambos decodificadores debe ser igual únicamente cuando se decodifica una cadena de bits continua. En el caso de que la cadena de bits de entrada se divida en tramas, habrá diferencias entre ambos.

Con la técnica truncamiento de trellis, si la cadena de entrada se divide en tramas, la capacidad de corrección de errores disminuye, [5] y [39]. Esto se debe a que los últimos bits de cada trama tienen menos protección contra errores, porque no se utiliza toda la memoria del decoding depth en su decodificación. Cuanto menores sean las tramas, más se acusa este inconveniente, por lo que la BERout aumenta. Es importante notar que esto no es un fallo, sino una característica del algoritmo.

Con la técnica cola de ceros, la BERout es independiente de si la decodificación es continua o está dividida en tramas. [40], [41].

Entonces para verificar que el decodificador UC3M es correcto debemos asegurar que cumple estas propiedades:

1. Si BERin es cero, BERout debe ser cero.

2. Si BERin>0, y la decodificación es continua, BERoutUC3M y BERoutXilinx deben converger hacia valores similares.

3. Si BERin>0 y la decodificación se divide en tramas. Entonces BERoutXilinx debe converger hacia un valor similar al obtenido en el punto 2, independientemente de la longitud de las tramas. Pero BEROutUC3M debe converger hacia un valor mayor. Además cuanto menores sean las tramas, más aumentará el valor de BEROutUC3M. [5], [39].

Para el proceso de medidas utilizamos el simulador de System Generator que instancia los dos decodificadores al mismo tiempo, y realizamos todos estos pasos:

1. Se fija un valor de BERinDecoder mediante la variable BER.

2. Se lanza la simulación.

3. BERout converge cuando se hayan alcanzado un mínimo de 1000 errores en el decodificador UC3M, en ese momento se para la simulación.

4. Almacenamos los valores BERoutUC3M y BERoutXilinx. Nos sirven para representar las gráficas BER frente a Eb/No.

5. Los pasos 1 al 4 se repiten para todos los niveles de BERinDecoder. Realizamos un total de 21 medidas entre los límites BERin=0,001 a BERin=0,1.

6. Los pasos 1 al 5 se repiten para los 6 decodificadores que hemos implementado. 5 UC3M, con decoding depth 24, 32, 36, 48 y 60. Y el de Opencores.

En el proceso se han necesitado más de 3000 horas de simulación. Pero se trata únicamente de horas de computación, no horas humanas de ingeniería. Porque únicamente debemos lanzar la simulación y dejar al PC trabajando el tiempo necesario hasta que converja. No es necesaria supervisión durante el tiempo de computación.

Sin embargo nuestros recursos son limitados, disponemos únicamente de un PC (Pentium dual core E2140 @ 1,6 GHz y 3 GB de RAM). Con esta máquina el tiempo en simular un millón de símbolos es de 6 horas. Debido a esto si la BERout es baja, no podemos esperar a que se alcancen los 1000 errores, puesto que se necesitarían varios días para ello. Entonces en algunos casos paramos la simulación cuando sólo se han alcanzado 100 errores, 20... Incluso algunas medidas no podemos realizarlas porque necesitaríamos muchísimo tiempo para alcanzar un número razonable de errores. Los valores cercanos a BERout = 10^{-6}, los medimos con poca precisión y los inferiores no podemos medirlos. Pero esto no es un problema porque disponemos de un rango de medidas más que suficiente. Aumentar el rango dispararía el tiempo de computación sin añadirnos apenas información útil.

En todas las medidas que hemos realizado hemos obtenido los resultados esperados, por lo que podemos asegurar que el decodificador UC3M es funcionalmente correcto.

Tercera prueba, las medidas deben coincidir con las del datasheet de Xilinx.

En la prueba anterior comparábamos las BERouts que obteníamos con nuestro simulador. Pero falta realizar una última comprobación, esas BERouts deben coincidir con las del datasheet de Xilinx, páginas 31 y 32 de [3]. La prueba nos permite verificar con un 100 % de seguridad, que realizamos el proceso de simulación y medidas correctamente.

En las *figuras 7.2 y 7.3* se puede ver esa coincidencia, por lo que damos por finalizado el proceso de verificación funcional.

7.7.2 Gráficas decodificador UC3M frente a Xilinx.

Resumimos todas las medidas en 8 gráficas. En ellas podremos comprobar los siguientes puntos:

1. Que el UC3M es funcionalmente igual al de Xilinx, siempre que la decodificación sea continua.

2. El efecto de utilizar una decodificación dividida en tramas. Más información en [5] y [39].

3. La variación en el rendimiento al variar decoding depth, [5], [6], [7], [37], [38] y [39].

4. Comparación de las medidas con las que aparecen en el datasheet de Xilinx, páginas 31 y 32 de [3].

5. Veremos cómo el decodificador de Opencores no es funcionalmente correcto.

Recordamos las características comunes en todos los decodificadores de las 8 figuras:
1. Constraint length K = 7. 64 estados, 6 registros de memoria.
2. Decode rate, R = 1/2.
3. Polinomio generador 171, 133 (octal). 1111001 y 1011011.

En las dos primeras gráficas, *figuras 7.2 y 7.3* mostramos todas las medidas del UC3M con decoding depth 36 y 48 y lo comparamos con el de Xilinx.

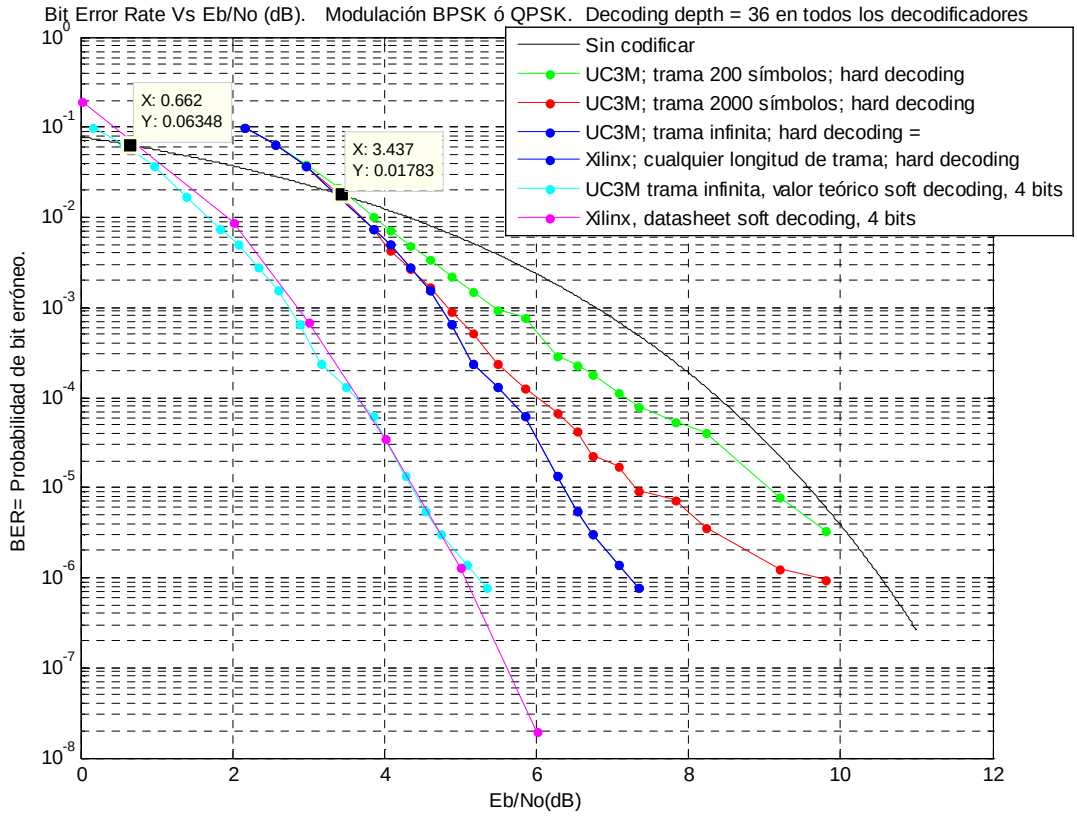


Figura 7.2: Comparativa decodificador UC3M y Xilinx. Decoding depth 36.

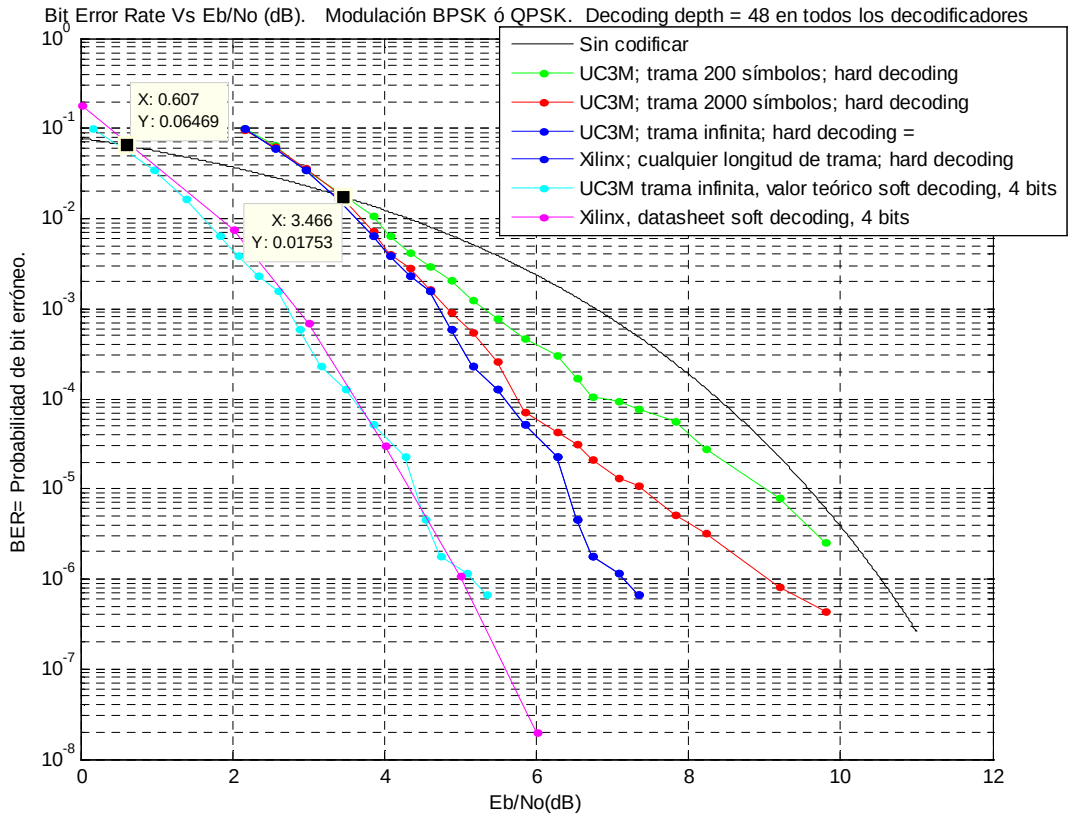


Figura 7.3: Comparativa decodificador UC3M y Xilinx. Decoding depth 48.

La información relevante de las gráficas anteriores es:

- En el simulador que empleamos, tanto el decodificador UC3M como el de Xilinx emplean decodificación dura. Las curvas azul, rojo y verde se obtienen a partir de los resultados de BERout que proporciona la simulación.

- El UC3M y el de Xilinx con trama de entrada continua comparten curva porque al simular hemos obtenido unos valores de BERout prácticamente idénticos, por lo que visualmente las dos curvas se superponen. Esto es lo que esperábamos y con ello se demuestra que el decodificador UC3M es funcionalmente igual al de Xilinx, y por tanto es correcto.

- Si la cadena de entrada se divide en tramas, el decodificador de Xilinx tiene el mismo rendimiento, pero el UC3M disminuye, [5], [39]. Esto también lo esperábamos porque en el UC3M empleamos truncamiento de trellis, mientras que en el de Xilinx se emplea cola de ceros [40], [41].

- Las medidas próximas a 10^{-6} no son precisas porque disponemos de recursos computacionales limitados. Obtenemos resultados aproximados pero suficientes para demostrar los objetivos que buscamos.

- En el datasheet de Xilinx no tenemos curvas con decodificación dura que nos permitan compararlas con las que hemos obtenido en las medidas. Entonces nos vemos obligados a usar curvas del datasheet con decodificación soft y compararlas con nuestras medidas. Esa curva es la rosa, que la obtenemos trasladando sus valores desde el datasheet, páginas 31 y 32 de [3].

- Los decodificadores que simulamos tienen decodificación dura, en ningún momento hemos implementado alguno con decodificación soft, por lo que no tenemos medidas con decodificación soft. Entonces la curva cian es teórica, sus valores los hemos obtenido sumando 2 dB, *nota 7.5*, de ganancia a la curva azul que es la que obtenemos simulando.

- La conclusión es que la curva teórica cian se aproxima mucho a la del datasheet, por lo que podemos asegurar que nuestras medidas y simulaciones son correctas. Se cumple la tercera prueba del apartado anterior. No se superponen exactamente porque nuestras medidas no son absolutamente precisas y porque el valor de 2 dB que sumamos lo obtenemos teóricamente.

- Pero estos dos últimos detalles no tienen ninguna importancia, y no merece la pena perder el tiempo en afinar más las medidas para que las curvas cian y rosa se superpongan. Tal y como están es más que suficiente para asegurar que nuestro decodificador se comporta correctamente.

Nota 7.5: Al utilizar una decodificación soft se obtiene una cierta ganancia frente a la dura. Esta ganancia no es constante para todos los valores de Eb/No pero puede aproximarse por una constante. En caso de utilizar 3 bits se obtiene una ganancia de 2dB respecto a la dura. Utilizar 4 bits no aumenta la ganancia de manera significativa frente a 3 bits, por lo que aproximamos la ganancia de 4 bits en 2 dB respecto a la decodificación dura. Explicación en [6], [7], [34], [35] y [36].

7.7.3 Efectos al variar el decoding depth.

Para conseguir el mejor rendimiento el decoding depth debe ser al menos 5 veces el constraint length = K, ver *apartado 2.5.3*. A partir de esta referencia si decoding depth es mayor que 5*K el rendimiento apenas mejora. Pero si es menor que 5*K, el rendimiento disminuye mucho. Para ampliar información consultar [5], [6], [7], [37], [38] y [39].

Las medidas de las siguientes gráficas demuestran que nuestro decodificador cumple estos fundamentos teóricos. En la *figura 7.4* se aprecia que utilizar un decoding depth de 60, (8.57 *K), proporciona unos 0.4 dB de ganancia frente al de 24, (3,42*K). Además también muestra como el UC3M se comporta exactamente igual que el de Xilinx si la cadena de entrada es continua, los dos comparten la misma curva porque sus medidas se superponen. Y si la cadena de entrada se divide en tramas, el de Xilinx se comporta igual, mientras que en el UC3M hay una disminución de rendimiento. Todo esto es una representación de lo que esperábamos teóricamente.

En las *figuras 7.5, 7.6 y 7.7* se aprecia como se obtienen prácticamente los mismos resultados utilizando decoding depth 32, (4,5*K), 36, 48 y 60. La ganancia que se obtiene al utilizar un valor mayor de 32 es muy pequeña, y no podemos medirla con exactitud porque nuestras curvas no tienen tanta precisión.

Sin embargo si se muestra con claridad como el rendimiento empeora al utilizar 24, así que se cumple lo que esperábamos teóricamente. Una disminución clara para una memoria < 5*K y una ganancia inapreciable para una memoria > 5*K.

En nuestro caso concreto la opción óptima es la de decoding depth 36, porque utilizar 48 o más apenas mejora la BER. Pero si se nota en recursos hardware, ya que se necesita el doble de memoria, ver *tabla 7.4*. La mejor opción dependerá del hardware sobre el que se descargue el código que hemos desarrollado. En el caso de usar una Virtex 4, lo mejor es usar 36 porque el tamaño máximo de palabra de sus bloques de memoria es de 36 bits.

En las figuras también se muestra como el decodificador de Opencores no funciona, su BER siempre está muy por encima del UC3M y el de Xilinx.

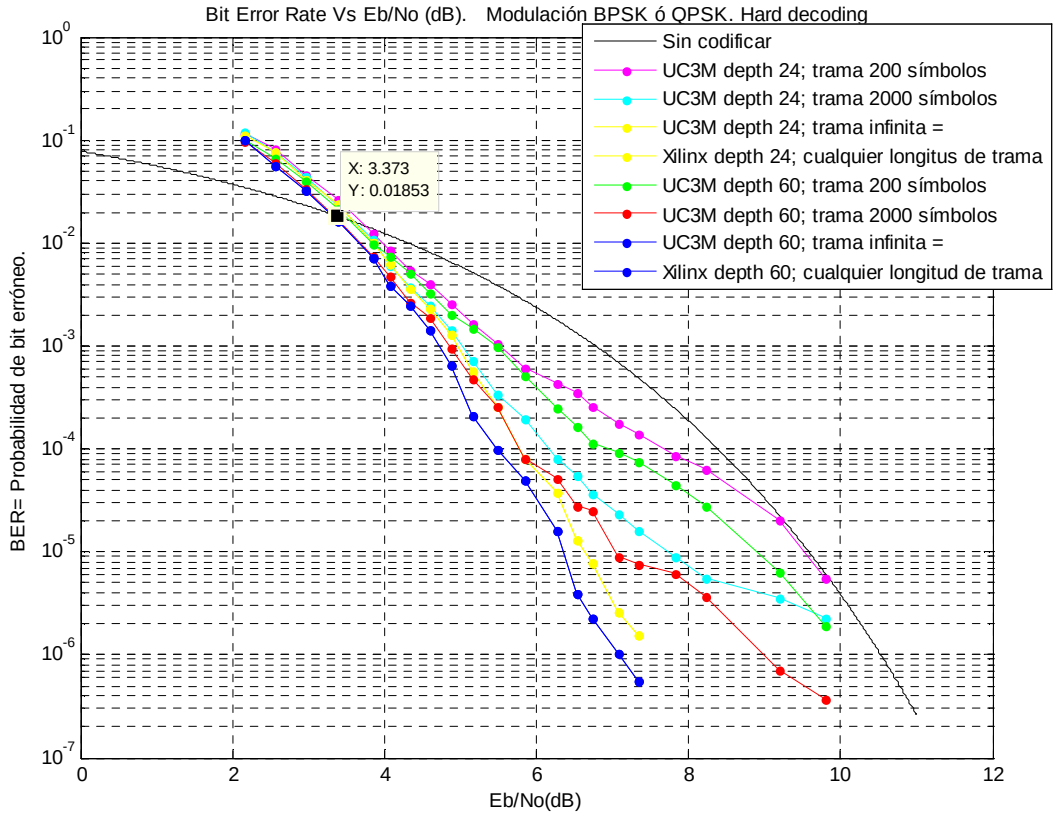


Figura 7.4: Comparativa decoding depth 60 frente a 24.

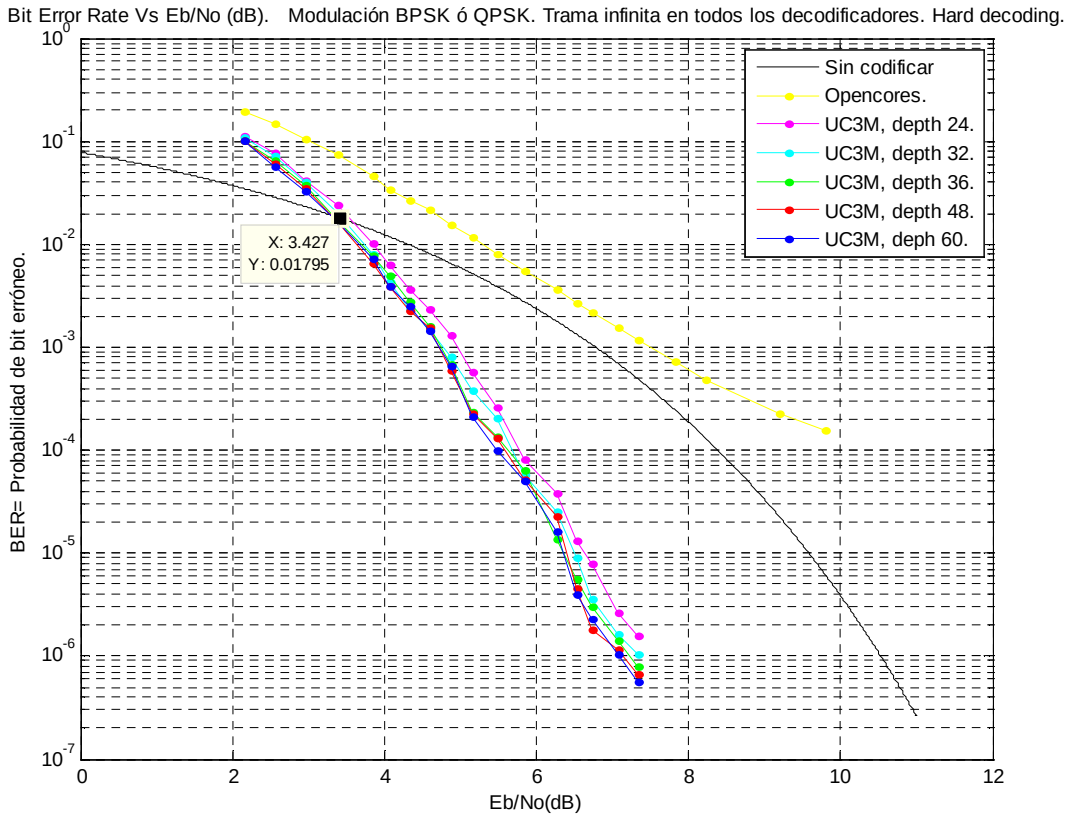


Figura 7.5: Variación en la BER al modificar decoding depth. Trama continua.

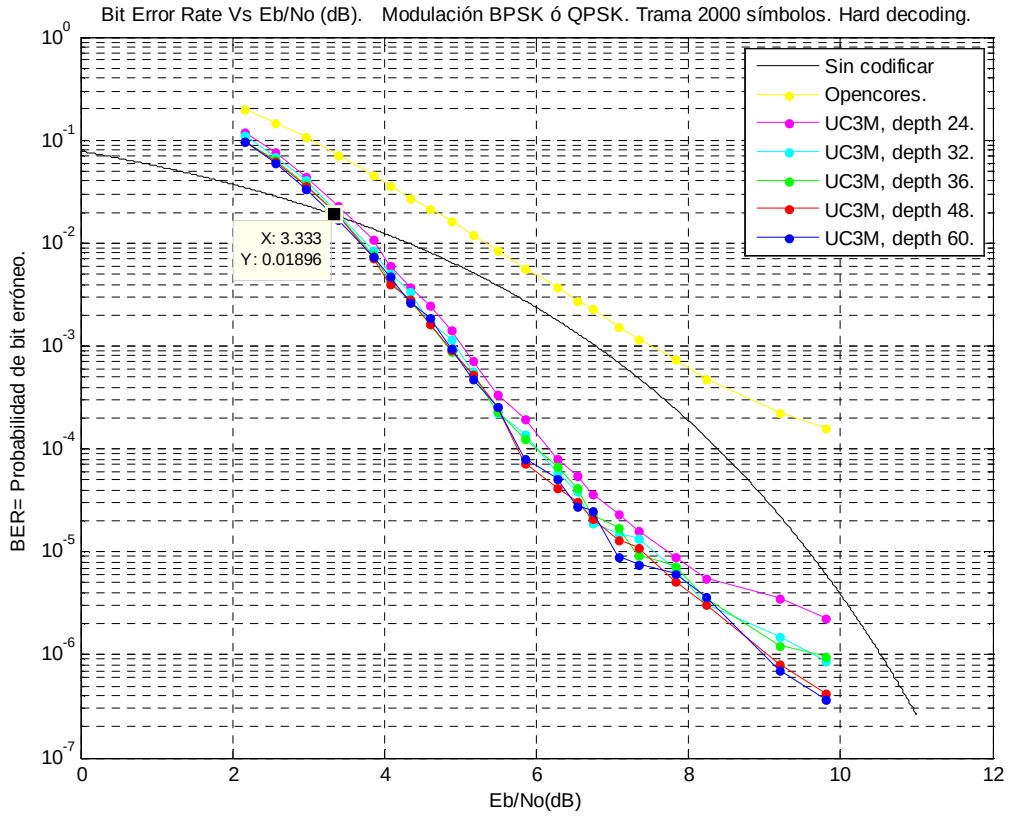


Figura 7.6: Variación en la BER al modificar decoding depth. Trama 2000 símbolos.

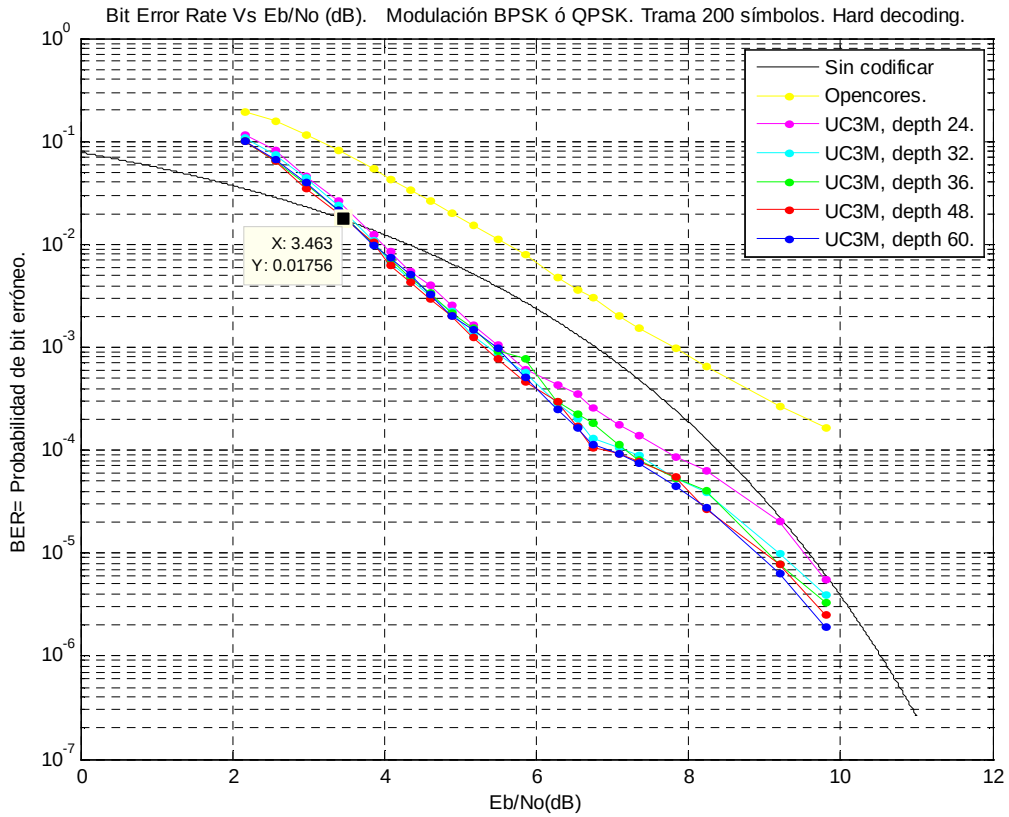


Figura 7.7: Variación en la BER al modificar decoding depth. Trama 200 símbolos.

7.7.4 Dependencia con la longitud de la trama.

En las figuras anteriores observamos que la BEROut del UC3M depende de la longitud de las tramas de entrada, mientras que la del Xilinx no. Consultar [5] y [39]. A continuación mostramos algunas medidas concretas:

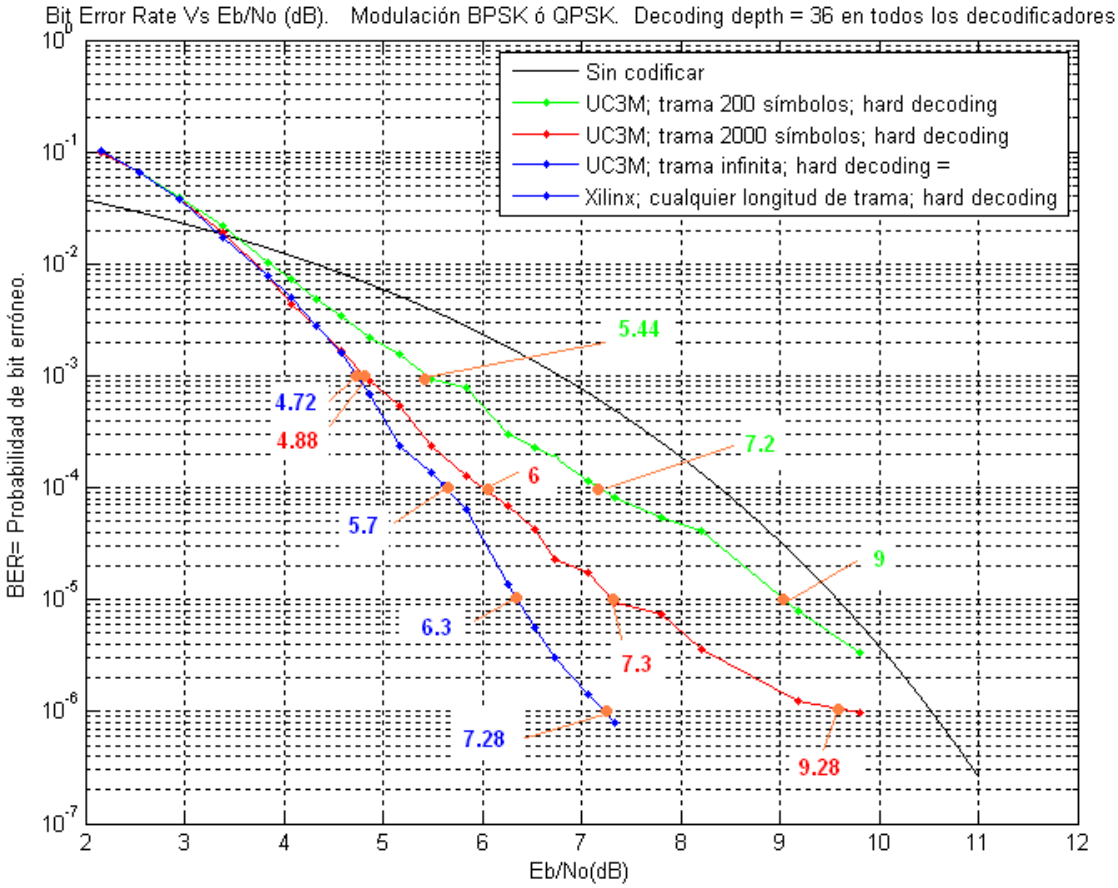


Figura 7.8: Pérdidas de Eb/No al disminuir el tamaño de la trama de entrada.

En el UC3M empleamos truncamiento de trellis, que consiste en iniciar la decodificación en el estado cero y a partir de ese estado ir rellenando el trellis con los símbolos de entrada. Cuando llegue el último símbolo se dejan de introducir símbolos en el trellis. En los *apartados 2.6.3 y 2.7.3* vimos como se realiza esta decodificación. También se ve el inconveniente de esta técnica, que consiste en que los últimos bits de cada trama tienen menos protección contra errores, porque están almacenados en un trellis con menos de decoding depth posiciones. Entonces la BERout depende de la longitud de la trama, cuánto menor sea la trama, mayor será el aumento de BERout.

En el Xilinx se emplea cola de ceros, que consiste en añadir m ceros al final de cada trama de entrada al codificador. Siendo m el número de registros del codificador, K-1. En esta ocasión la decodificación comienza en el estado cero y finaliza en el estado cero, puesto que la cola de ceros fuerza a los símbolos de la malla trellis a finalizar en ese estado. Esto tiene la ventaja de que todos los símbolos de la trama tienen la misma protección contra errores, porque en todos ellos se utiliza un trellis con decoding depth posiciones. Por eso la BERout no depende de la longitud de la trama.

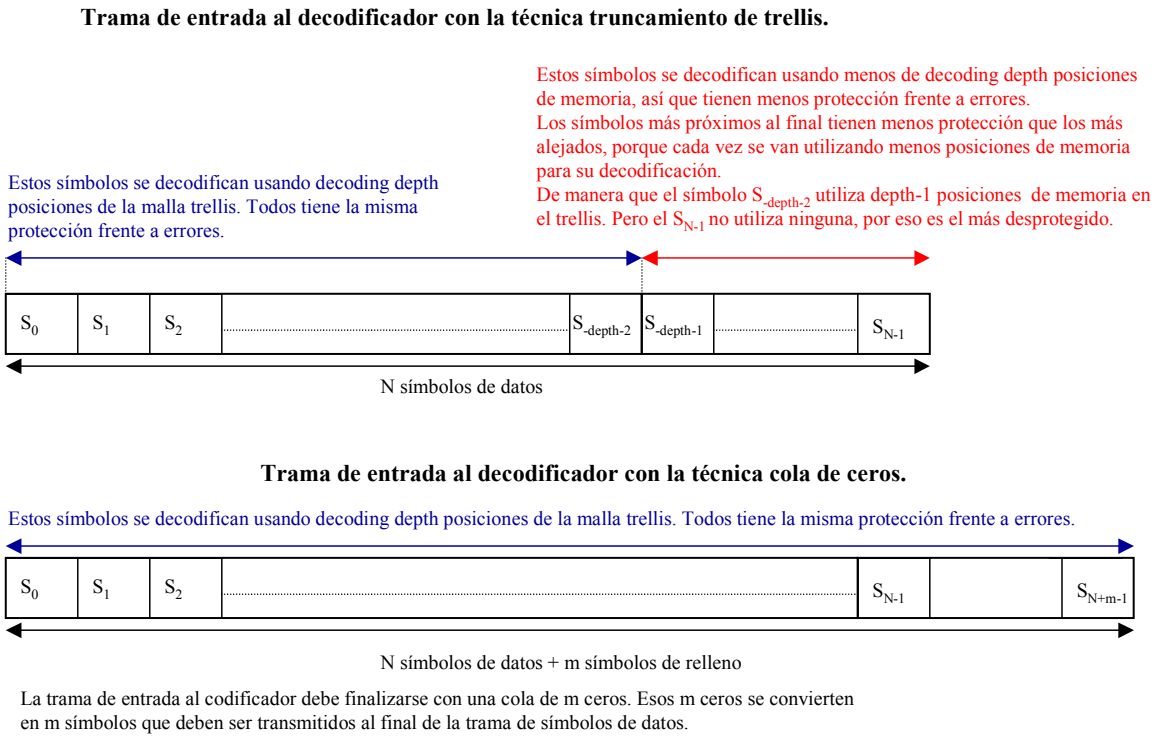


Figura 7.9: Técnicas truncamiento de trellis y cola de ceros.

Las ventajas e inconvenientes de ambas técnicas son:

- La técnica truncamiento de trellis es la más simple, lo que permite que el diseño del decodificador sea más sencillo. La ventaja de esto es que el código es más simple lo que repercutirá en menor área al sintetizar.

- Con la técnica truncamiento de trellis en las tramas hay únicamente bits de datos. Mientras que con la cola de ceros, cada trama de entrada al codificador hay que finalizarla con una cola de m ceros. El inconveniente de esto, es que habrá que transmitir m símbolos redundantes por cada trama. Por tanto el sistema truncamiento de trellis es más eficiente, puesto que se transmiten N símbolos por trama, mientras en el otro N+m.

La técnica más habitual es la cola de ceros porque aporta la ventaja de que no se pierde ganancia al dividir la codificación en tramas. Sin embargo la otra también se emplea porque en algunos sistemas es más importante la simplicidad.

Además si en el sistema se van a emplear cadenas de entrada continuas, la mejor opción es la truncamiento de trellis. Porque en estas condiciones el rendimiento es igual a la cola de ceros, pero conserva su mayor simplicidad. Esto es lo que ocurre en nuestro diseño, nuestras especificaciones son las de un decodificador que trabaje con tramas continuas, entonces con estas condiciones la mejor opción es truncamiento de trellis.

Referencias sobre cola de ceros y truncamiento de trellis en [40] y [41].

7.7.5 Gráficas decodificador UC3M frente a Opencores.

Ya hemos descrito que el Opencores tiene fallos de síntesis y por eso no lo empleamos.

Otro inconveniente es que hay que añadir 36 bits de relleno al final de cada trama de entrada al codificador, ver *apartado 4.4*. En cambio en el de Xilinx este número se reduce a 6 y en el UC3M a cero.

Pero a los fallos anteriores hay que sumarle el más importante de todos, consiste en que tiene errores funcionales y lo hemos visto en las gráficas anteriores. A continuación mostramos la gráfica definitiva en la que lo comparamos con un decodificador UC3M y otro de Xilinx.

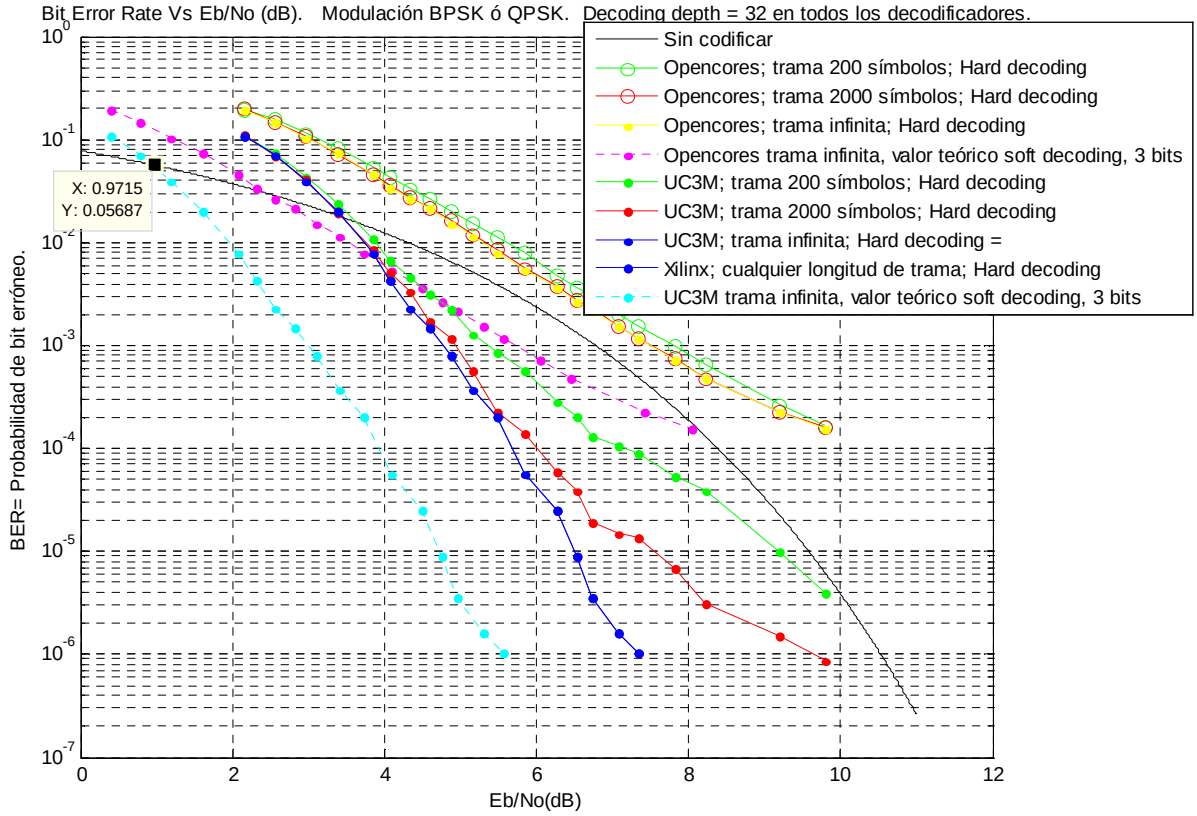


Figura 7.10: Rendimiento Opencores.

La información más relevante es:

- El decodificador Opencores no se comporta como el de Xilinx, así que no es funcionalmente correcto.

- Además ni siquiera proporciona ganancia frente al sistema sin codificar. Al contrario, la BERout aumenta respecto al sistema sin codificar. Por lo tanto no tiene ningún sentido utilizar el decodificador de Opencores. Por que se aumenta la complejidad del sistema, y a cambio no sólo no se mejoran sus prestaciones, sino que empeoran.

- Hemos buscado un motivo lógico que permita utilizarlo y lo hemos encontrado. Consiste en emplear decodificación soft. En este caso se obtendría la curva rosa, que proporciona una ligera ganancia respecto al sistema sin codificar. Más información sobre decodificación soft en [6], [7], [34], [35] y [36].

- Esta curva rosa es una representación teórica, la obtenemos sumando 2 dB a la curva Opencores trama infinita, que es la que obtenemos con nuestras medidas.

- Utilizando decodificación soft podría justificarse el empleo de este decodificador, ya que mejora algo respecto al sistema sin codificar. Pero vemos que aún en ese caso sigue siendo peor que un UC3M con decodificación dura. Y mucho peor que el valor teórico que obtendríamos con el UC3M empleando decodificación soft, curva cian.

- De nuevo la curva cian es teórica. La obtenemos sumando 2 dB a la curva que obtenemos simulando un UC3M con trama continua.

- Por último se aprecia que en Opencores la BERout depende del tamaño de la trama, pero esta dependencia no es tan acusada como en el UC3M. Pero este detalle carece de importancia, porque en ninguna ocasión se obtiene un rendimiento equiparable al UC3M.

7.8 Comportamiento ante errores de ráfaga.

En todas las gráficas y medidas anteriores el ruido es blanco y gaussiano, canal AWGN. Esto produce unos errores uniformemente distribuidos en los bits de entrada al decodificador. Y es en estas condiciones con las que un sistema de codificación convolucional y decodificación Viterbi funciona correctamente, disminuyendo la BER del sistema.

Sin embargo en un canal real los errores en los bits recibidos se producen de manera aleatoria pero no uniforme. Lo habitual es que los errores se concentren en ráfagas. Pero si a la entrada del decodificador llegasen errores de ráfaga, éste no funcionaría de la manera deseada y no sería capaz de reducir la BER. Por eso en todos los sistemas FEC, forward error correction, que emplean codificación convolucional, es obligatoria la presencia de un entrelazador y un desentrelazador. Este dispositivo baraja los bits, convirtiendo los errores de ráfaga en uniformemente distribuidos. Ver *figura 6.1 y 6.3*.

El mal funcionamiento de la codificación convolucional con errores de ráfaga es un hecho demostrado en teoría y que puede consultarse en la bibliografía, [42], [43], [44] y [45]. Sin embargo nosotros como valor añadido al proyecto hemos medido la BERout en condiciones de error de ráfaga. De esta manera observando la siguiente figura comprobamos como lo que dice la teoría es cierto.

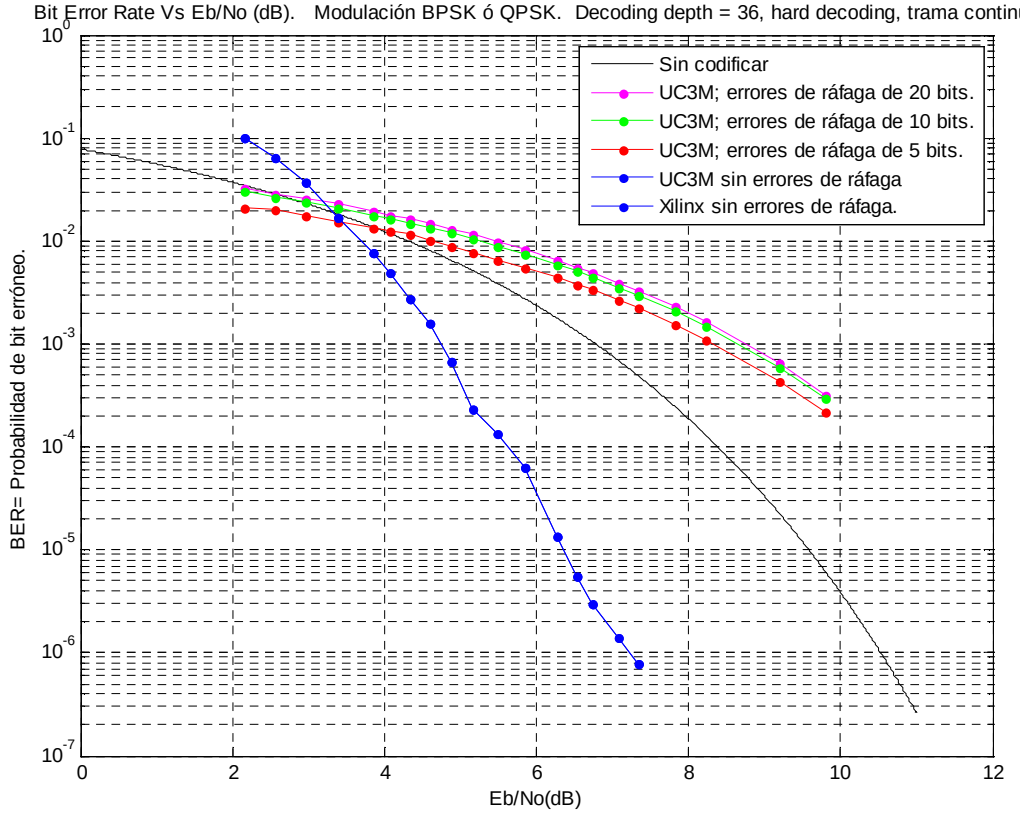


Figura 7.11: BER frente a Eb/No con errores de ráfaga.

Condiciones de la medida:

Simulamos solamente el de decoding depth 36 y trama continua. Con esto es suficiente para demostrar que el rendimiento es muy inferior al que se obtiene sin errores de ráfaga.

Sólo hay errores de ráfaga, en el intervalo entre ráfagas no se produce ningún error. La estructura de la cadena de entrada se representa en la siguiente figura:

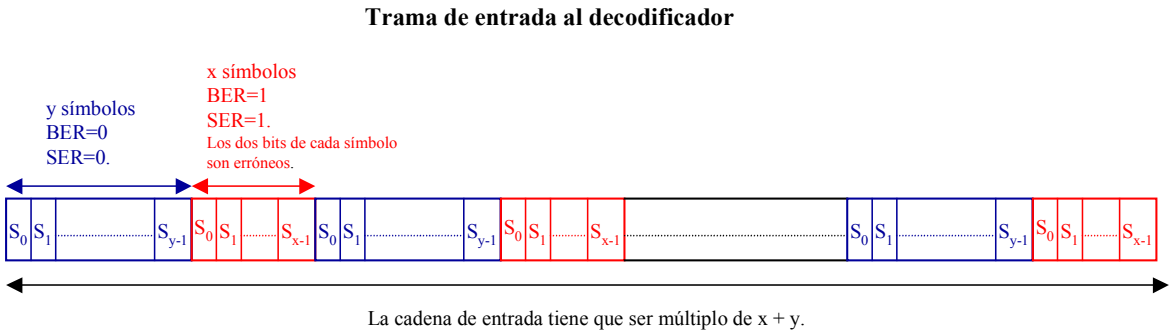


Figura 7.12: Formato de la cadena de entrada con errores de ráfaga.

Cada ráfaga tiene x símbolos y entre dos ráfagas consecutivas hay un intervalo de y símbolos sin errores. Así conseguimos medir el efecto de únicamente los errores de ráfaga, sin que haya otras variables que afecten a la medida.

Hemos diseñado los simuladores de Xilinx de manera que permitan simular con facilidad los errores de ráfaga. Pero debemos tener en cuenta que en esta ocasión la constante BER del simulador valdrá siempre cero. Esta constante determina el número de errores uniformemente distribuidos que habrá en la entrada. En esta ocasión es cero, porque en el intervalo entre ráfagas no hay ningún error.

De manera que calculamos el valor de BERinDecoder de esta manera:

$$BERinDecoder = \frac{x}{x+y}$$

Antes de realizar las simulaciones debemos rellenar las columnas 1, 2, 3 y 4 de la *tabla 7.10*. Las dos primeras columnas son iguales que en la *tabla 6.1*, la 3 es fija a 5, 10 ó 20. Sólo hay que calcular la cuarta, mediante esta ecuación:

$$y = \frac{x}{BERinDecoder} - x$$

Con las columnas 3 y 4 tenemos los parámetros necesarios para la simulación. Así que sólo falta lanzar la simulación y cuando converja rellenar la columna 5.

Las curvas BER frente a Eb/No se forman con los puntos de las columnas 1 y 5.

Los parámetros en cada una de las simulaciones son:

1. BER = 0,0. Porque no queremos errores en el intervalo entre ráfagas.

2. NumBitsSimulacion = NumBitsPorTrama = NumRafagas*(x+y). La cadena de entrada es continua, pero sólo podemos simular un número finito de símbolos, que debe ser múltiplo de x + y. El valor de NumRafagas debe ser el suficiente para que la simulación converja. Esto ocurrirá cuando se produzcan al menos 1000 errores en la salida del decodificador.

3. ErrorRafaga = '1'.

4. MaxIntervaloRafaga = MinIntervaloRafaga = y.

5. MaxLongRafaga = MinLongRafaga = x. En el simulador podemos emplear con facilidad tanto ráfagas como intervalos entre ráfagas de longitud variable. Sin embargo no lo necesitamos en estas medidas, y siempre las realizaremos con longitudes fijas.

Tabla 7.10: Puntos para representar una curva con errores de ráfaga.				
$Eb_{SinCodificar}$/No dB	BERinDecoder	Ráfaga x símbolos	Intervalo entre ráfagas Y símbolos	BerOutDecoderUC3M Depth = 36, trama continua
2,154	0,1	10	90	$3,06 * 10^{-2}$
2,5465	0,09	10	101	$2,7 * 10^{-2}$
2,953	0,08	10	115	$2,34 * 10^{-2}$
3,38	0,07	10	133	$2,097 * 10^{-2}$
3,8325	0,06	10	157	$1,794 * 10^{-2}$
3,072	0,055	10	172	$1,647 * 10^{-2}$
4,323	0,05	10	190	$1,5 * 10^{-2}$
4,585	0,045	10	212	$1,35 * 10^{-2}$
4,864	0,04	10	240	$1,2 * 10^{-2}$
5,162	0,035	10	276	$1,047 * 10^{-2}$
5,485	0,03	10	323	$9 * 10^{-3}$
5,845	0,025	10	390	$7,5 * 10^{-3}$
6,251	0,02	10	490	$6 * 10^{-3}$
6,528	0,017	10	578	$5,1 * 10^{-3}$
6,731	0,015	10	657	$4,485 * 10^{-3}$
7,072	0,012	10	823	$3,6 * 10^{-3}$
7,336	0,01	10	990	$3 * 10^{-3}$
7,809	0,007	10	1419	$2,088 * 10^{-3}$
8,218	0,005	10	1990	$1,5 * 10^{-3}$
9,181	0,002	10	4990	$6,03 * 10^{-4}$
9,8	0,001	10	990	$3,03 * 10^{-4}$

7.9 Referencias

Proporcionamos el enlace Web siempre que exista, accedemos a estos enlaces por última vez en noviembre de 2011. Además tenemos una copia de seguridad en el CD del proyecto de toda la documentación sin copyright.

[1] Página Web de Xilinx: http://www.xilinx.com/

[2] Página Web decodificador Viterbi Xilinx:
http://www.xilinx.com/products/intellectual-property/Viterbi_Decoder.htm

[3] Xilinx "Viterbi Decoder v7.0 Data Sheet", ds247.pdf, 1 de Marzo de 2011:
http://www.xilinx.com/support/documentation/ip_documentation/viterbi_ds247.pdf

[4] Xilinx "LogiCORE IP Viterbi Decoder v7.0 User Guide", ug745.pdf, 18 Octubre 2010:
http://www.xilinx.com/support/documentation/ip_documentation/ug745_viterbi_decoder.pdf

[5] Todd K.Moon. "Error Correction Coding. Mathematical Methods and Algorithms".Wiley-Interscience, pp. 487-490, 2005.

[6] Shu Lin and Daniel J. Costello, Jr. "Error Control Coding. Fundamentals and Applications." Prentice-Hall Series in Computer Applications in Electrical Engineering, pp. 343-345, 1983.

[7] John G. Proakis. "Digital Communications". McGraw Hill, pp. 506-511, fourth edition 2001.

[8] Página Web tarjeta del laboratorio Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 XC4VSX55 10FF1148:
http://www.lyrtech.com/products/vhs-adc.php#

[9] Specification sheet, tarjeta del laboratorio Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 XC4VSX55 10FF1148:
http://www.lyrtech.com/documents/product_sheets/Specification%20sheet%20-%20VHS-ADC%20%28hi_res%29.pdf

[10] Página Web de la Xilinx Virtex-4:
http://www.xilinx.com/support/index.htm#nav=sd-nav-link-19224&tab=tab-sd

[11] Xilinx "Virtex-4 Family Overview", ds112.pdf, v3.1, 30 Agosto 2010:
http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf

[12] Xilinx "Virtex-4 FPGA User Guide", ug070.pdf, v2.6, 1 Diciembre 2008:
http://www.xilinx.com/support/documentation/user_guides/ug070.pdf

[13] Miguel Ángel Freire Rubio. "Diseño Síncrono de Circuitos Digitales". Ingeniería Técnica de Telecomunicación, EUITT, Universidad Politécnica de Madrid, septiembre 2008.
http://www.euitt.upm.es/uploaded/464/DS_Sep_2008.pdf

[14] "Electrónica Digital, Tema 3, Diseño Síncrono". Departamento de sistemas electrónicos y de control, EUITT, Universidad Politécnica de Madrid, Curso 2010-2011.
http://www.euitt.upm.es/uploaded/464/teoria/tema3/Tema3.pdf

[15] Delmar Thomson. "Digital Design with Cpld Applications and VHDL". Thomson Delmar Learning, pp 275-298, 1ª edition June 28, 2000.

[16] Enoch O. Hwang. "Digital Logic and Microproccesor Design with VHDL". Brooks Cole, pp. 175-182, 2005.

[17] Documentación disponible en la ayuda de Xilinx Ise:
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/isehelp_start.htm

http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_c_configuration_overview.htm

http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_c_using_xst_for_synthesis.htm

[18] Documento Xilinx: "Synthesis and Simulation Design Guide". UG626 (v 13.3) October 19, 2011
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/sim.pdf

[19] Documento Xilinx: "ISE In-Depth Tutorial". UG695 (v13.3) October 19, 2011
http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_3/ise_tutorial_ug695.pdf

[20] Herbert Dawid, Olaf J. Joeressen and Heinrich Meyr. "Chapter 17 Viterbi Decoders: High Performance Algorithms and Architectures". Páginas 17-26.
http://www.eecs.berkeley.edu/newton/Classes/EE290sp99/lectures/ee290aSp996_1/vit_chap17.pdf

[21] Mohammed Benaissa and Yiqun zhu."A Novel High-Speed Configurable Viterbi Decoder for Broadband Access". EURASIP Journal on Applied Signal Processing,volumen 13, pp. 1317–1327, 2003 Hindawi Publishing Corporation.
http://www.hindawi.com/journals/asp/2003/865460/abs/

http://downloads.hindawi.com/journals/asp/2003/865460.pdf

[22] Sh. Sanjay Sharma and Pushpinder Kaur. "Implementation of Low Power Viterbi Decoder on FPGA". Pp. 12, 34-42, Junio 2006.
http://dspace.thapar.edu:8080/dspace/bitstream/123456789/292/1/r8044117.pdf

[23] Andres Iborra y Juan Suardiaz. "Unidad 4 Dispositivos Lógicos Programables". Universidad Politécnica de Cartagena, Febrero 2003.
http://www.dte.upct.es/personal/andres_iborra/docencia/sis_elec/pdfs/t4.pdf

[24] Uwe Meyer-Baese. "Digital Signal Processing with Field Programmable Gate Arrays". Springer, pp. 3-27, 2001.

[25] Stephen Brown and Zvonko Vranesic. "Fundamentals of Digital Logic with VHDL Design". Mc Graw Hill, pp. 1-6 y 899-916, second edition 2005.

[26] Luis Jacobo Álvarez Ruiz de Ojeda. "Historia de los Circuitos Digitales Configurables". Universidad de Vigo.
http://www.dte.uvigo.es/logica_programable/documentos/curso_disenho_digital_con_CDCs/Documento_historia_PLDs.pdf

[27] Christian B. Schlegel and Lance C. Pérez. "Trellis and Turbo Coding". IEEE Press Series on Digital & Mobile Communication, pp. 117-120, 2004.

[28] Bernard Skalar. "Digital Communications, Fundamentals and Applications". Prentice Hall PTR , pp. 408-419, segunda edición 21 Enero 2001.

[29] Especificación decodificador Viterbi Opencores:
Mikael Johnson. "Specification of Viterbi HDL Code Generator". 27 Mayo de 2004.
La documentación y el código del decodificador Viterbi de Opencores están accesibles en la Web de Opencores. El decodificador Viterbi está en la categoría arithmetic core.
Sitio Web Opencores: http://opencores.org/
Sitio Web Viterbi Opencores. http://opencores.org/project,vhcg

[30] Distribuidor de Xilinx en España, empresa Silica:
Gabriel Cutillas, Field Application Engineer.Gabriel.Cutillas@silica.com
http://www.silica.com/

[31] Documento Xilinx: "Xilinx INC. Core License Agreement", Enero 2009:
http://www.xilinx.com/ipcenter/doc/xilinx_click_core_site_license.pdf

[32] Xilinx "Question and Answers. SignOnce IP License Agreement", 9 Agosto 2007:
http://www.xilinx.com/ipcenter/CLA_QA.pdf

[33] Ian Kuon and Jonathan Rose. "Measuring the Gap between FPGAs and ASICs". The Edward S. Rogers Sr. Department of Electrical and Computer Engineering University of Toronto. 22 Febrero 2006.
http://www.eecg.toronto.edu/~jayar/pubs/kuon/kuonfpga06.pdf
{ikuon, jayar}@eecg.utoronto.ca

[34] Jorge Castiñeira Moreira and Patrick Guy Farrell. "Essentials of Error-Control Coding". Wiley, pp 189-196, 2006.

[35] Bernard Skalar. "Digital Communications, Fundamentals and Applications". Prentice Hall PTR , pp. 396-399, segunda edición 21 Enero 2001.

[36] Khmaies Ouahada. "Viterbi Decoding of Ternary Line Codes". Publicado por VDM Verlag Dr. Müller, pp 35-80, 2008.

[37] Khmaies Ouahada. "Viterbi Decoding of Ternary Line Codes". Publicado por VDM Verlag Dr. Müller, pp 81-105, 2008.

[38] Krishna Sankar. "Viterbi with finite survivor state memory". 27 julio de 2009.
http://www.dsplog.com/2009/07/27/viterbi-with-finite-survivor-state-memory/

[39] "Codificación de Canal". Universidad Técnica Federico Santa María, UTFSM, páginas 29 y 30.
http://www2.elo.utfsm.cl/~elo341/ComDig09_HC.pdf

[40] Michael Francis. "Viterbi Decoder Block Decoding – Trellis Termination and Tail Biting". Documentación de Xilinx, XAPP551 (v2.0), 30 de Julio de 2010.
http://www.xilinx.com/support/documentation/application_notes/xapp551.pdf

[41] Christian B. Schlegel and Lance C. Pérez. "Trellis and Turbo Coding". IEEE Press Series on Digital & Mobile Communication, pp. 138-141, 2004.

[42] John G. Proakis. "Digital Communications". McGraw Hill, pp. 468-470; 717-724; 830-832. Fourth edition 2001.

[43] Bernard Skalar. "Digital Communications, Fundamentals and Applications". Prentice Hall PTR , pp. 461-469, segunda edición 21 Enero 2001

[44] Peter Sweeny. "Error control Coding. From Theory to Practice". Wiley 2002, pp 22-23; 56-57; 64; 220-221; 232-235. Publicado en 2002.

[45] Krishna R. Narayanan and Gordon L. Stüber. "Performance of Trellis-Coded CPM with Iterative Demodulation and Decoding". IEEE Transactions on Communications, Vol 49, Nº4, Abril 2011.
http://www.tamu.edu/faculty/commtheory/papers/trelliscodedCPM_comm01.pdf

[46] Frank Vahid. "Digital Design with RTL Design, Verilog and VHDL". John Wiley and Sons Publishers, pp 377-380, 2011.

[47] "Pipelining". ECEN 6263 Advanced VLSI Design, 7 Noviembre 2006. http://lgjohn.okstate.edu/5263/lectures/pipe.pdf

[48] Suryanarayana B. Tatapudi and José G. Delgado-Frias. "Designing Pipelined Systems with a Clock Period Approaching Pipeline Register Delay". School of Electrical Engineering and Computer Science. Washington State University
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.3074&rep=rep1&type=pdf.

{statapud, jdelgado}@eecs.wsu.edu

CAPÍTULO 8

8. INTEGRACIÓN ViterbiDecoder.vhd EN EL SISTEMA WIMAX.

8.1 Objetivos

En el *apartado 1.3* describimos los pasos seguidos en el proyecto. En esta etapa ya hemos finalizado correctamente los pasos 1 al 9 y procedemos a realizar el 10:

1. Realizar un estudio teórico previo, en el que adquirimos los conocimientos necesarios para realizar el proyecto.

2. Diseñar un codificador convolucional y un decodificador Viterbi con VHDL.

3. Optimizarlos en área y velocidad.

4. Sintetizarlos e implementarlos en hardware.

5. Desarrollar un simulador con VHDL.

6. Verificar el codificador y el decodificador mediante simulación funcional y post place & route.

7. Integrar el codificador y el decodificador en System Generator.

8. Desarrollar un simulador con System Generator.

9. Mediante el simulador del punto anterior, verificamos el codificador y el decodificador tras integrarlos en System Generator. Además comparamos nuestro decodificador con el Xilinx IP core Viterbi decoder.

10. Integrar el decodificador en el prototipo WiMAX.

11. Elaborar esta memoria, describiendo: el trabajo realizado; los conocimientos adquiridos; y proporcionando las referencias bibliográficas más adecuadas para cada uno de los aspectos relacionados con este proyecto.

En el *apartado 8.3* describimos las tareas que hay que realizar para cumplir con el hito número 10. Se trata de tareas muy sencillas, porque nuestro decodificador es estándar e independiente de la tecnología. De manera que puede integrarse en cualquier sistema y en cualquier hardware sin ningún problema.

8.2 Características sistema WiMAX.

El sistema WiMAX original constituye un proyecto fin de carrera diferente: "Prototipado de un Sistema WiMAX MIMO 2x2". Realizado también en la Universidad Carlos III de Madrid por David Díaz Martín y Roberto Prieto Alonso. Referencias [2] y [3].

El archivo principal se denomina:
"sistcomp_BPSK_1trama_comp3_simu_madeinrx_2008_08_23.mdl".

Las especificaciones de este proyecto WiMAX son:

- Estándar IEEE 802.16d-2004 (WiMAX fijo). Especificación completa en [1].

- MIMO 2x2 OFDM, multiplexado por división en frecuencias ortogonales en la capa física. Modulación de datos BPSK, (modulación binaria por salto de fase). Las siglas MIMO (*multiple input - multiple output*), indican que hay más de una antena transmisora y receptora. Lógicamente, 2x2 indica que hay 2 antenas en el transmisor y otras dos en el receptor.

A continuación hacemos un breve resumen de las características genéricas de WiMAX. No nos extendemos porque citamos una amplia bibliografía en el *apartado 1.9.3*.

- Son las siglas de Worldwide Interoperability for Microwave Access. (Interoperabilidad mundial para acceso por microondas).

- La familia de estándares WiMAX define un acceso a redes inalámbricas de banda ancha. En esta definición se incluyen la capa MAC, (Control de Acceso al Medio), y la capa PHY (Capa Física).

- El WiMAX consiste en un nodo central que se comunica de manera inalámbrica con múltiples usuarios, consiguiendo velocidades similares a las de la tecnología ADSL y fibra óptica.

- La aplicación más común consiste en dar servicios de banda ancha en zonas con baja densidad de población, o de difícil acceso, donde no exista una red de xDSL ni de fibra óptica. En este escenario tender una red supone un coste por usuario muy elevado. De manera que realizando una conexión inalámbrica mediante WiMAX se consigue una calidad de servicio similar a un coste muy inferior.

- En sistemas WiMAX con visión directa entre el nodo central y la antena del usuario se permiten alcanzar distancias superiores a 50 Km. Son sistemas LOS (Line of Sight).

- En los sistemas sin visión directa entre el nodo central y la antena del usuario se permiten celdas de varios kilómetros de radio. Son sistemas NLOS (Non Line of Sight). Esto es típico en entornos urbanos, donde hay múltiples obstáculos: por ejemplo árboles, edificios... Además la antena del usuario se ubica en: tejados, terrazas, azoteas..., en vez de en lugares más apropiados como colinas o torres.

8.3 Integración de ViterbiDecoder.vhd en el sistema WiMAX.

El sistema WiMAX original realizado por David Díaz Martín y Roberto Prieto Alonso incluye un decodificador Viterbi, implementado con un IP core de Xilinx, Viterbi decoder v5.0. Con estas características:

- Constraint length K=7. 2^{K-1}= 64 estados.
- Decode rate R = ½, (tasa = 1/2). 2 bits de entrada, 1 de salida.
- Polinomio generador 171, 133 (octal). 1111001 y 1011011.
- Codificación dura.
- No sistemático y no recursivo.

El decodificador que hemos implementado, *ViterbiDecoder.vhd*, tiene las mismas características. Por tanto podemos sustituir el IP core de Xilinx por nuestro módulo.

Nuestro decodificador, al igual que los decodificadores comerciales, implementa un algoritmo Viterbi exacto. Así que todos los dispositivos se reducen a una caja negra que contiene el mismo algoritmo. Por tanto se puede sustituir cualquier dispositivo comercial por nuestro módulo y el funcionamiento será el mismo, la capacidad de corrección de errores será igual (*nota 8.1).* Lógicamente para que la sustitución sea posible, todas las características que hemos detallado en la lista anterior deben ser iguales en los decodificadores.

No existirá ningún problema de compatibilidad con el hardware sobre el que se implemente el sistema, porque *ViterbiDecoder.vhd* es completamente multiplataforma. Esto es un valor añadido fundamental, porque nuestro módulo es compatible con cualquier soporte hardware programable, y puede sustituir a cualquier decodificador Viterbi comercial. Que esté implementado en una FPGA de cualquier fabricante y también en el resto de dispositivos lógicos programables, por ejemplo los más comunes: ASIC, CPLD, PLD...

Volvemos a insistir en un aspecto importante: nuestro decodificador no es específico para la aplicación de WiMAX. Se trata de un decodificador genérico, que puede emplearse en cualquier protocolo de comunicaciones en el que se necesite decodificar un código convolucional (2,1,K=7); no sistemático; no recursivo; codificación dura y función de transferencia: $G(x) = [1+x^2+x^3+x^5+x^6 \quad 1+x+x^2+x^3+x^6]$.

Nota 8.1: Como vimos en los *apartados 2.6.1 y 7.7.1*, el algoritmo Viterbi tiene una cierta componente de decisión aleatoria. Esto implica que ante la misma cadena de entrada, la cadena de salida obtenida por un decodificador puede ser diferente a la obtenida por otro. Puede haber diferencias instantáneas en los bits de salida de ambos, pero esto no afecta al rendimiento, porque la capacidad de corrección de errores de los dos decodificadores converge al mismo valor. Esto no se trata en ningún caso de un error, sino que es una característica del algoritmo. Por este motivo, un módulo Viterbi puede sustituirse por otro aunque se obtengan resultados instantáneos diferentes en la salida.

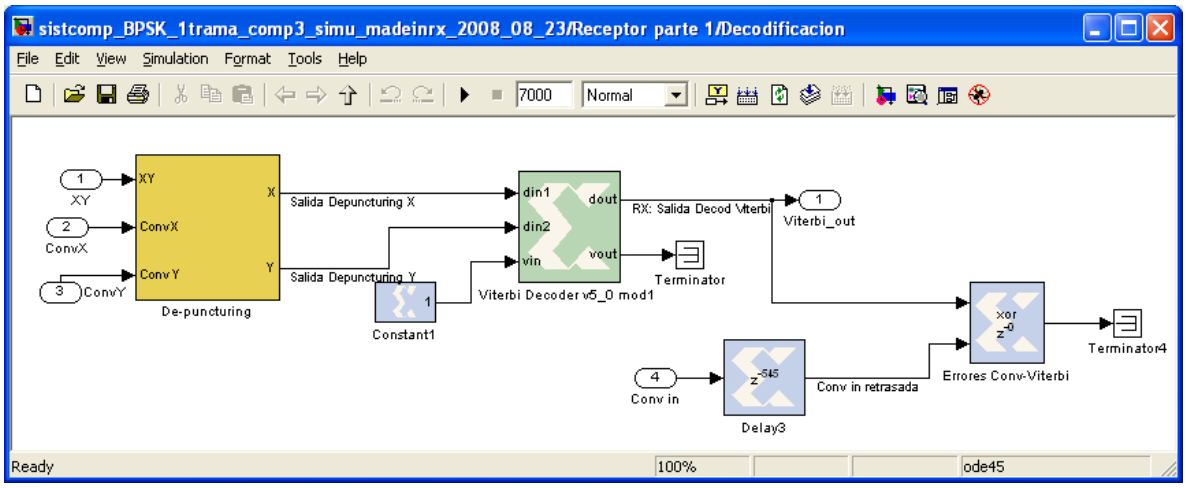


Figura 8.1: Xilinx IPcore Viterbi decoder v5.0 en el sistema original.

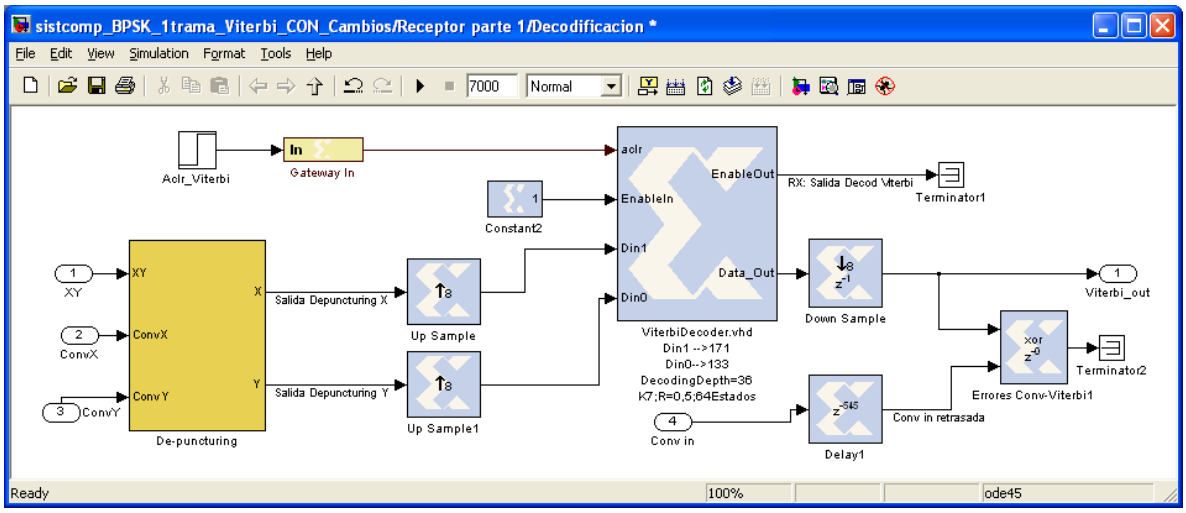


Figura 8.2: Sustitución del IP core de Xilinx por ViterbiDecoder.vhd.

Para utilizar a pleno rendimiento el Viterbi de Xilinx en una FPGA compatible de Xilinx, es necesario comprar una licencia, full license key.

También se puede adquirir una licencia de evaluación gratuita, full system hardware evaluation license key , válida durante 120 días. Esta licencia permite realizar todo el proceso de síntesis y simulación. También permite descargar el código en la tarjeta, pero el código sólo funcionará en la tarjeta durante un tiempo limitado.

Nosotros no disponemos de la licencia completa ni de la de evaluación. No las necesitamos porque disponemos de nuestro módulo, *ViterbiDecoder.vhd*, que podemos utilizar gratuitamente y funciona igual, porque ambos implementan el mismo algoritmo.

Nota 8.2: No podemos sintetizar el WiMAX original porque aparece el siguiente error: "You must install the full license or evaluation license of this block to realize it in hardware." El error se refiere al Viterbi decoder v5.0 y es lo que esperábamos. Tan sólo nos indica lo que ya sabíamos, que no disponemos de la licencia.

El siguiente paso es sustituir el IP de Xilinx por el *ViterbiDecoder.vhd*. Al hacer esto no hay que hacer ningún cambio ni añadido en el *Viterbidecoder.vhd*, porque lo hemos diseñado multiplataforma, así que no hay ningún problema de compatibilidad.

En el WiMAX hay que hacer unas pequeñas modificaciones para adaptar las interfaces y las velocidades de reloj del nuevo módulo. Los cambios, ver *figura 8.2*, son:

Hay que acceder al submódulo: "Receptor parte1/Decodificacion"

1. Se quita el Viterbi decoder v5_0 y se pone en su lugar una black box que en su interior contiene el *Viterbidecoder.vhd*.

2. Din1 corresponde al polinomio 171 y Din0 al 133.

3. El IP core de Xilinx emplea un T_{CLK} para decodificar un dato, y a su entrada llega un dato cada T_{CLK} segundos. Por tanto trabaja con frecuencia F_{CLK} y la frecuencia de llegada de datos es también F_{CLK}. Sin embargo *ViterbiDecoder.vhd* necesita 8 T_{CLKs} para decodificar un dato. Entonces si la frecuencia con la que llegan los datos a la entrada es F_{CLK}, el decodificador debe tener una frecuencia de reloj = $8F_{CLK}$. Por eso hay que añadir bloques Up Sample en la entrada del módulo, que multiplican la frecuencia de reloj por 8, y Down Sample en la salida, que dividen la frecuencia de reloj por 8. Gracias a estos bloques el sistema WiMAX trabaja con F_{CLK} y *ViterbiDecoder.vhd* con $8F_{CLK}$.

4. Es obligatorio marcar la casilla copy samples en el módulo Up Sample, porque cada dato debe estar estable en la entrada del Viterbi durante 8 $T_{CLKs.}$

5. EnableIn siempre estará activo, Constant2 = 1. Debido a esta circunstancia el decodificador trabajará en modo continuo.

6. Es necesario añadir una señal externa de $\overline{aclr}$.

8.4 Referencias.

[1]Norma IEEE Std 802.16™ –2004. "IEEE Standard for Local and metropolitan area networks. Part 16: Air Interface for Fixed Broadband Wireless Access Systems". 1 Octubre 2004.
http://wirelessafrica.meraka.org.za/wiki/images/8/83/802.16-2004.pdf

[2]David Díaz Martín. "Prototipado de un Sistema WiMAX MIMO 2x2". Proyecto Fin de Carrera en Ingeniería Técnica Superior de Telecomunicación. Universidad Carlos III de Madrid. Enero 2010.

[3]David Díaz Martín y Roberto Prieto Alonso. "Estudio Práctico sobre el prototipado de un sistema MIMO 2x2 para WiMAX" . Estudio Tecnológico en Ingeniería Técnica Superior de Telecomunicación. Universidad Carlos III de Madrid. 24 Enero 2008.

CAPÍTULO 9

9. PRESUPUESTO.

9.1 Objetivos.

El objetivo de este capítulo es realizar una estimación del coste del proyecto. Para ello lo primero que hay que definir es qué hemos hecho y qué productos se podrían vender.

Hemos realizado el código y la documentación de un codificador convolucional, un decodificador Viterbi y simuladores. Con el valor añadido de que todo el código es multiplataforma. Estos productos pueden venderse en el mercado. La situación sería la de un cliente que necesita implementar cualquier sistema en hardware, y ese sistema precisa de un módulo decodificador Viterbi, un codificador convolucional, o necesita simular sus bloques. Nosotros le entregaríamos el código de nuestros módulos y el cliente los implementaría en su sistema y su hardware.

Todo el coste del proyecto es en: diseño y desarrollo del código y la documentación, más simulación y verificación de los prototipos realizados. Por tanto un aspecto importante, es que una vez finalizado el diseño y las pruebas de los prototipos, el coste de fabricación de nuestros productos es cero. Porque únicamente entregaríamos al cliente un CD con el código y la documentación. El soporte hardware sobre el que se sintetiza el código lo tendría el cliente y no tiene relación con nuestros módulos. Esta es la mayor ventaja de nuestros diseños frente a los de la competencia, que pueden implementarse sobre cualquier soporte hardware: cualquier FPGA de cualquier fabricante y también en el resto de dispositivos programables: ASICs, CPLDs, PLDs ...

En el mercado existen productos similares, pero la mayoría son dependientes de la tecnología. Por ejemplo: Xilinx, Actel, Lattice y Altera disponen de IP cores que implementan un decodificador Viterbi, pero sólo son válidos para determinadas FPGAs de su propio catálogo de productos. Disponemos de todas las referencias de estos cores en el *apartado 1.9.5* y en [70] a [77] del *tema 1*.

Para realizar un estudio completo contactamos con Gabriel Cutillas, distribuidor de Xilinx en España a través de la empresa Silica. Referencia [46] del *tema 1*. Amablemente nos proporciona toda la información que necesitamos:

- El precio del IP core Viterbi decoder de Xilinx es de $5000 (site license) y de $15000 (worldwide license). Evidentemente se trata de licencias anuales. Para tener derecho a recibir las actualizaciones, es preciso renovar dichas licencias en la fecha correspondiente, con costes de $1000 y $3000, respectivamente.

- Este IP no tiene royalties. Esto significa que una vez adquirida la licencia se puede instanciar en todos los dispositivos que se necesiten. La licencia cuesta lo mismo si se va a fabricar una tarjeta o un millón. (Un millón es un simple ejemplo, no hay ningún límite máximo).

- A nivel de mercado, los fuentes VHDL no suelen entregarse, salvo acuerdos particulares y a precios bastante superiores. Lo normal es entregar netlists reubicables. Por tanto con el precio de $5000 o $15000 jamás se entregaría el código fuente, únicamente las netlist reubicables.

- El IP convolutional encoder no requiere licencia. Puede utilizarse libremente de manera gratuita.

De la información anterior extraemos una ventaja adicional de nuestro decodificador. Consiste en que el IP core de Xilinx necesita actualizaciones con el tiempo, mientras que el nuestro no. Esto se debe a que continuamente se están desarrollando nuevas FPGAs, y cuando se adquiere el decodificador de Xilinx, está optimizado para instanciarse en las FPGAs más modernas disponibles en ese momento. Pero en el futuro saldrán al mercado FPGAs y tarjetas más modernas, y el IP de Xilinx no estará optimizado para ellas, o incluso no podrá instanciarse en ellas. De manera que se hace necesario renovar las licencias anualmente, con el coste indicado de $1000 ó $3000.

Lo habitual es que un cliente quiera utilizar los dispositivos hardware actuales, es poco habitual realizar nuevos diseños sobre tarjetas y FPGAs antiguas. Por tanto este es un punto a favor de nuestro desarrollo, porque al ser completamente multiplataforma, se puede instanciar y está optimizado para las tarjetas actuales y también para las futuras. Para ello nos hemos valido de que VHDL es un lenguaje estándar, que aceptan los dispositivos actuales, futuros y pasados (incluso los descatalogados).

9.2 Elaboración presupuesto.

Volvemos a insistir en que todo el coste del proyecto es en el diseño de la primera unidad. Una vez finalizado el diseño de la primera unidad, las siguientes unidades tienen coste cero, porque únicamente entregaríamos al cliente un CD con el código y la documentación.

El presupuesto total del proyecto lo elaboramos teniendo en cuenta estos 3 aspectos:

1. El número de horas de ingeniería empleadas en el diseño.
2. Los gastos en materiales y hardware.
3. Los gastos en software.

9.2.1 Coste en horas de ingeniería.

El proyecto se realiza entre dos personas, un alumno, Miguel Viñé Viñuelas, con un nivel profesional equivalente al de un ingeniero junior. Y un tutor que dirige el proyecto, Enrique San Millán Heredia, con un nivel profesional equivalente al de director de proyecto.

No hemos tenido una dedicación exclusiva para este proyecto, sino que lo hemos ido realizando según la disponibilidad del tiempo. Así que el tiempo transcurrido entre el comienzo y el final del proyecto, no indica cuántas horas hemos empleado en realizarlo. Para poder obtener este dato lo que hemos hecho es apuntar diariamente las horas de trabajo dedicadas. De esta manera obtenemos un dato final de 1451+100 horas de trabajo, que reflejan con total exactitud el coste en horas del proyecto. 1451 corresponden a las empleadas por el alumno proyectante y 100 a las empleadas por el tutor que dirige el proyecto.

A continuación se muestran las tareas que hemos realizado y el número de horas empleadas en cada una.

Tabla 9.1: Fases del proyecto y horas dedicadas a las mismas.			
Concepto	Horas empleadas	Coste hora (€)	Coste total (€)
Búsqueda y lectura de documentación. Estudio teórico previo sobre codificación convolucional y decodificación Viterbi.	**364**	**48**	**17472**
Desarrollo código incluyendo comentarios. Código codificador, decoderverilog.v, ViterbiDecoder.vhd y todos los simuladores.	**568**	**48**	**27264**
Tiempo total empleado en la memoria:	**519**	**48**	**24912**
Documentación ViterbiDecoder.vhd (tema 5).	113	48	5424
Temas 1, 8, 9, 10 y anexos.	86	48	4128
Documentación decoderverilog.v (capítulo 4).	38	48	1824
Documentación codificador convolucional (tema 3).	27	48	1296
Documentación simuladores (capítulo 6).	74	48	3552
Documentación resultados de la simulación (tema 7).	72	48	3456
Estudio tecnológico (tema 2).	109	48	5232
Tiempo total empleado por el alumno proyectante	**1451**	**48**	**69648**
Tiempo empleado por el director del proyecto.	**100**	**70**	**7000**
Simulación	Más de 3000	0	0
Coste total en recursos humanos	**1551**	**1451*48 + 100*70**	**76648**

Consideraciones:

a) No comenzamos a desarrollar el código hasta haber estudiado en profundidad los aspectos que necesitamos conocer sobre la codificación convolucional y la decodificación Viterbi. En esta primera fase del proyecto empleamos 364 horas.

b) Para el proceso de simulación se han necesitado más de 3000 horas, que constituyen el tiempo de computación empleado en las simulaciones. Este tiempo no tiene coste, porque para simular sólo hay que escribir los parámetros en el simulador, lanzar la simulación y esperar hasta que estén los resultados. Pero durante la espera no se necesita ninguna supervisión, y por tanto el coste de estas más de 3000 horas es cero. Sin embargo sí que es necesario dedicar un tiempo para escribir los resultados en el simulador, lanzar la simulación y trasladar los resultados a las tablas y figuras que aparecen en el *tema 7*. Este tiempo está incluido en las 72 horas que empleamos en documentar el *tema 7*.

c) El coste de una hora de trabajo lo estimamos realizando un estudio de mercado y comparando los honorarios de una hora de ingeniería en diferentes proyectos. Los honorarios medios que observamos son: 48 euros para la hora de trabajo de un ingeniero junior y 70 horas en el caso del director de proyecto.

d) Hasta el año 2008 no era necesario realizar un estudio de mercado para estimar el coste de la hora de ingeniería. Porque estos costes estaban publicados en la lista de honorarios orientativos del COIT, (Colegio Oficial de Ingenieros de Telecomunicación, referencia disponible en [78] del *tema 1*). Sin embargo esto ya no es posible, porque por modificación de la Ley de Colegios Profesionales, mediante Ley 25/2009 de 22 de diciembre, no es posible seguir publicando estas listas. El Colegio no puede elaborar baremos de honorarios, ni siquiera orientativos, salvo que sean con la finalidad de tasar costas en los procedimientos judiciales. En [80] del *tema 1*, disponemos de la ley completa publicada en el BOE. Y en [79] del *capítulo 1*, disponemos de la normativa del COIT, que explica los cambios en el Colegio que son obligados por Ley.

9.2.2 Costes materiales.

Tabla 9.2: Gastos en material.

Equipo	Precio nuevo (€)	Precio estimado tras finalizar el proyecto (€)	Coste imputable al proyecto (€)
Tarjeta de desarrollo Lyrtech VHS-ADC con FPGA Xilinx Virtex 4 xc4vsx55 10ff1148	18000	15000	3000
PC Pentium Dual Core E2140 2 GB RAM	400	100	300
Monitor 17 pulgadas	150	50	100
Impresora HP Deskjet 2300	70	20	50
Pen drive 16 GB	30	10	20
Mantenimiento PC	150		150
Material oficina e impresión de documentos			80
Conexión ADSL	19,95/mes		50
Lugar de trabajo	0 (Porque este gasto está incluido en los honorarios de 48 ó 70 € por hora de ingeniería)		0
Comida trabajadores			0
Total			**3750**

Consideraciones:

a) La tarjeta de desarrollo Lyrtech VHS-ADC se comparte entre dos departamentos: Tecnología Electrónica y Teoría de la Señal y Comunicaciones. La tarjeta no se compró exclusivamente para nuestro proyecto, sino que ha servido de base para la realización de múltiples proyectos. Y se puede reutilizar para proyectos futuros. Por estos motivos estimamos un coste imputable a nuestro proyecto de 3000 € sobre los 18000 € que costó inicialmente.

b) El hardware: PC, monitor, impresora y pen drive pueden utilizarse en el futuro, así que debemos calcular la amortización que se ha hecho de ellos para estimar el coste imputable al proyecto. Para realizar este cálculo podemos aplicar fórmulas que relacionan directamente la amortización de los equipos por hora de uso. Sin embargo estas fórmulas no son de utilidad en este caso. Porque no disponemos de ninguna estimación de la vida útil del equipo en el momento de su compra, tampoco conocemos la cantidad de horas que han estado en funcionamiento, ni podemos estimar la vida útil que les resta a los equipos.

c) Por todos los motivos que exponemos anteriormente, consideramos que la mejor manera de determinar la amortización del producto es mediante su devaluación. Para ello obtenemos la diferencia entre el equipo nuevo, al inicio del proyecto, y el valor residual que tiene el equipo en el mercado de segunda mano, en el momento de finalizar el proyecto.

d) Utilizamos una conexión ADSL, pero en ningún momento hacemos uso completo de ella. La conexión es más importante en la fase de búsqueda de documentación, mientras que en las fases de desarrollo del código y elaboración de la memoria, la conexión ADSL apenas se utiliza. Por este motivo no se puede imputar el coste total de 19,95 euros al mes al proyecto. Consideramos que 50 euros es un coste apropiado y cubre el uso que le hemos dado.

9.2.3 Costes software.

Tabla 9.3: Costes software

Concepto	Precio licencia (€)	Coste imputable al proyecto (€)
Xilinx ISE 8.1 service pack 3. Incluye Ise Simulator.	Gratuita	0
Matlab 7.1 (release 14) service pack 3. Incluye Simulink	3000	100
System Generator for DSP 8.1.01. Incluye el simulador.	6000	200
Software Lyrtech	0 (Sin coste porque la licencia viene incluida en los 18000 € de la tarjeta Lyrtech VHS-ADC)	0
Windows XP	0 (Sin coste porque la licencia viene incluida en el PC Pentium Dual Core de 400 €)	0
Microsoft office 97	100	30
Total		**330**

- El departamento de Tecnología Electrónica dispone de una licencia de Matlab que incluye Simulink. Esta licencia permite usar Matlab y Simulink en varios PCs del departamento, con un coste imputable de 100 euros por PC. Para nuestro proyecto utilizamos un único PC, Intel Dual Core, y por tanto el coste imputable es de 100 euros. Referencias de Matlab en [17], [18] y [20] del *capítulo 1*.

- El departamento de Tecnología Electrónica dispone de una licencia de System Generator for DSP, que cuesta unos 6000 euros. De nuevo puede utilizarse en varios PCs del departamento, con un coste imputable de 200 euros por PC. Así que el coste para nuestro proyecto es de 200 €. Documentación sobre System Generator en el *apartado 1.9.2*.

9.3 Conclusiones.

El presupuesto total de nuestro proyecto se obtiene sumando los recursos humanos, los gastos materiales y los costes software = 76648+3750+330 = 80728 €.

Como hemos visto en el *apartado 9.1*, la licencia del decodificador de Xilinx cuesta $5000 (site license) y $15000 (worldwide license). Además se necesita una renovación anual, con costes de $1000 y $3000, respectivamente.

Entonces una primera conclusión es que el coste de desarrollo es mayor que el coste de la licencia. Esto es algo lógico, lo único que indica es que si el objetivo del proyecto fuese vender nuestro producto, para ser viable económicamente sería necesario vender más de una unidad. Sucede exactamente lo mismo con la empresa Xilinx, porque el presupuesto de desarrollo del decodificador será mucho mayor que el coste de la licencia. Como es lógico su negocio consiste en realizar un desarrollo pensando en vender más de una unidad.

Otra consideración es que además del código hemos realizado un estudio tecnológico: del codificador convolucional, decodificador Viterbi, simuladores y sistemas FEC. Proporcionamos una documentación detallada sobre todos estos temas, y además seleccionamos la bibliografía necesaria donde se puede ampliar información. Este trabajo es necesario, porque la documentación que hemos desarrollado puede servir como punto de partida para proyectos futuros realizados en la Universidad Carlos III.

Una última conclusión, es que si el objetivo del proyecto hubiese sido solamente desarrollar un código para venderlo, no hubiese sido necesario realizar el estudio tecnológico del *tema 2*. Por lo que el presupuesto se habría reducido en 109 horas *48 €/hora= 5232 €.

El presupuesto total de este proyecto asciende a la cantidad de 80728 euros.

Leganés a 8 de mayo de 2012
El ingeniero proyectista

Firmado: Miguel Viñé Viñuelas

CAPÍTULO 10

10. CONCLUSIONES.

10.1 Diseño y verificación de ViterbiDecoder y EncoderK7.

Hemos cumplido el objetivo principal de: diseñar, implementar, simular y verificar un decodificador Viterbi en hardware, con código VHDL. Sus características básicas son: K=7, R=1/2, polinomio 1111001-1011011. Se trata del módulo final del proyecto, (*ViterbiDecoder.vhd*), que integramos en el prototipo WiMAX.

También hemos diseñado, implementado, simulado y verificado un codificador convolucional (2, 1, K=7), en hardware con código VHDL, al que denominamos. *EncoderK7.vhd*. Este bloque es necesario para realizar el modelo del sistema FEC con el que simulamos el decodificador.

En todo el código que hemos desarrollado en este proyecto hemos cumplido con todos los pasos requeridos en un diseño hardware:

1. Se cumplen todas las reglas de diseño síncrono.
2. Optimizamos el código, obteniendo el mejor compromiso entre mínima área ocupada y máxima frecuencia de reloj.
3. Hemos verificado que no hay errores estructurales ni de diseño.
4. Verificamos que el circuito puede implementarse en hardware sin ningún error.
5. Obtenemos el archivo de configuración de la FPGA (*.bit).

Además añadimos una ventaja a todos los módulos que desarrollamos con VHDL: *ViterbiDecoder.vhd*, *EncoderK7.vhd* y los simuladores. Consiste en que los realizamos completamente multiplataforma, independientes de la tecnología. Por tanto, el codificador y el decodificador pueden implementarse en cualquier FPGA de cualquier fabricante. Y también en el resto de dispositivos lógicos programables, por ejemplo los más comunes: ASICs, CPLDs, PLDs... Además, como el simulador codificado con VHDL también es multiplataforma, se puede realizar la simulación y verificación del codificador y el decodificador con cualquier herramienta de síntesis.

Lógicamente, las condiciones anteriores se cumplen siempre que la herramienta de síntesis y el hardware acepten VHDL IEEE 1076-1993. Esto es lo habitual y ocurrirá en la gran mayoría de ocasiones, porque VHDL es un lenguaje estándar en la industria, aceptado por la gran mayoría de fabricantes y de dispositivos lógicos programables.

La característica de multiplataforma es una ventaja respecto a la mayoría de los decodificadores Viterbi y codificadores convolucionales existentes. Porque lo habitual es que estos dispositivos sean dependientes de la tecnología, de manera que sólo se pueden sintetizar en determinados soportes hardware.

Hemos diseñado simuladores, codificados con VHDL y System Generator, que modelan con total exactitud el entorno real que tendrán tanto el codificador como el decodificador, cuando se implementen en una placa hardware real. Se trata de un modelo de un sistema FEC, en el que se incluyen todas las variables que influirán en su comportamiento cuando se implementen en un sistema real. Estas variables son:

- Ruido uniforme con amplitud variable.
- Errores de ráfaga. Con longitud e intervalo entre ráfagas variable.

- Secuencia de datos aleatorios de entrada al codificador. Distribuidos mediante tramas de longitud y espacio entre tramas variable, o mediante una secuencia continua (trama de longitud infinita). Canal BSC, (canal binario simétrico).

- Profundidad de memoria del decodificador (decoding depth).

Para verificar que tanto el decodificador como el codificador son funcionalmente correctos, deben cumplirse estas 3 condiciones:

1. En un sistema FEC ideal, sin ruido, el código debe ser reversible. Esto se cumple cuando en nuestro simulador se obtiene BERoutDecoder = 0, con estas condiciones de entrada: amplitud del generador de ruido = 0, por tanto BERinDecoder = 0.

2. El decodificador y el codificador deben comportarse igual que cualquier dispositivo que implemente un codificador convolucional y un decodificador Viterbi. Para realizar esta comprobación, comparamos nuestros módulos con el Xilinx IP core Viterbi decoder v5.0, y con el Xilinx IP core convolutional encoder v3.0. Se aplican las mismas señales de entrada a nuestro módulo y al IP core de Xilinx correspondiente. A continuación se comparan los resultados en sus salidas, mostrando los resultados en tiempo real. *Nota 10.1.*

3. El rendimiento, (capacidad de corrección de errores), de nuestro decodificador, debe ser igual al del Xilinx IP core Viterbi decoder v5.0. Para comprobarlo, representamos las gráficas de rendimiento de *ViterbiDecoder.vhd* en términos BER frente a Eb/No. Y nos aseguramos de que coinciden con las gráficas de rendimiento mostradas en el datasheet del core de Xilinx.

La comparación de nuestros módulos con los cores de Xilinx, sólo está disponible en los simuladores codificados con System Generator. No hemos incluido esta opción en los codificados con VHDL, porque entonces dejarían de ser multiplataforma. Este es el motivo principal por el que hemos desarrollado dos simuladores que modelan el mismo sistema FEC, con las mismas señales y variables, utilizando dos lenguajes HDL diferentes. Además, en el modelo con System Generator, aprovechamos las ventajas que ofrece Simulink para mejorar la interfaz gráfica y facilitar el manejo del simulador.

Mediante la simulación se obtiene el rendimiento del decodificador en términos BERoutDecoder frente a BERinDecoder. A continuación, representamos el rendimiento en gráficas BER frente a Eb/No, particularizadas para este sistema: ruido AWGN, canal BSC y modulación BPSK ó QPSK. Verificamos que las medidas que hemos obtenido se correspondan en todo momento con los resultados teóricos esperados, cumpliendo los 3 puntos anteriores. Las conclusiones más importantes tras analizar los resultados son:

a) Las gráficas BER frente a Eb/No se corresponden exactamente con las correspondientes al Xilinx IP core Viterbi decoder v5.0, mostradas en su datasheet.

b) El comportamiento de nuestro codificador y decodificador, es igual al de los módulos Xilinx IP core convolutional encoder y Viterbi decoder, que usamos como referencia. *Nota 10.1.*

c) Representamos el comportamiento del decodificador frente a todas las variables que influyen en su rendimiento. Estas variables son:
   - Cubrimos todo el rango de Eb/No.
   - Longitud de la trama de entrada = 200, 2000 símbolos y trama continua.
   - Profundidad de memoria del decodificador = 24, 32, 36, 48 y 60.

d) Demostramos que es necesaria la presencia de un intercalador-deintercalador, para convertir los errores de ráfaga en errores uniformemente distribuidos. La ausencia de estos bloques en el sistema FEC disminuye significativamente el rendimiento.

Nota 10.1: Ante una misma secuencia de entrada, la secuencia de salida de ambos codificadores debe ser exactamente igual. Sin embargo, la secuencia de salida de ambos decodificadores no tiene porque coincidir. Esto no es un error, sino una característica de la decodificación Viterbi. El algoritmo tiene una componente aleatoria, debido a la cual las salidas instantáneas de dos decodificadores pueden ser diferentes. Lo que debe cumplirse, es que cuando la secuencia de entrada alcanza una longitud suficiente, ambas salidas deben converger al mismo valor de BERoutDecoder.

Por tanto, tras realizar el proceso completo de: diseñar, implementar, simular y verificar un decodificador Viterbi en hardware, obtenemos las siguientes conclusiones finales:

El proceso de diseño hardware se ha realizado correctamente. Con el valor añadido de que todo el código VHDL que hemos desarrollado es multiplataforma.

Realizamos una simulación completa tanto del decodificador como del codificador. Cubriendo todas las combinaciones posibles en las señales y variables que influyen en su comportamiento. Y verificando que se cumplen todas las condiciones necesarias. **Los resultados han sido siempre los esperados, por lo que verificamos que hemos desarrollado un codificador y un decodificador funcionalmente correctos. Implementan exactamente la función matemática de un decodificador Viterbi (K=7, R=1/2, 1111001-1011011) y un codificador convolucional (2, 1, K=7).**

La función matemática tanto del decodificador como del codificador está especificada y es la misma para todos los dispositivos. De manera que, **nuestros módulos tienen el mismo funcionamiento y los mismos puertos de entrada-salida, que el resto de decodificadores y codificadores existentes ajustados a las mismas especificaciones.** Hemos verificado esta propiedad, comparando nuestros bloques con el Xilinx IP core Viterbi decoder y el Xilinx IP core convolutional encoder.

Integramos el decodificador Viterbi en el prototipo WiMAX. El proceso consiste simplemente en instanciar ViterbiDecoder.vhd, añadiendo los bloques necesarios para adaptar su frecuencia de reloj a la del sistema WiMAX.
Además se trata de un decodificador genérico, que puede emplearse, sin cambios ni añadidos, en cualquier aplicación en la que se necesite decodificar un código convolucional.

El simulador realizado con VHDL es genérico y multiplataforma. Esto nos permite simular cualquier decodificador Viterbi ajustado a las mismas especificaciones, y empleando cualquier herramienta de simulación.

10.2 Características ViterbiDecoder.vhd

Al implementar el decodificador hemos cumplido con las especificaciones básicas, indicadas en el *apartado 1.4*. Además hemos ampliado esas especificaciones iniciales, añadiendo mejoras al decodificador.

Las especificaciones básicas que hemos cumplido son:

- Constraint length K = 7; $2^{K-1} = 64$ estados.
- Decode rate R = ½, (tasa = 1/2). 2 bits de entrada, 1 de salida.
- Polinomio generador 171, 133 (octal). 1111001 y 1011011.
- Decodificación dura.
- 6 puertos de entrada: Din1, Din0, EnableIn, $\overline{aclr}$, ce, clk.
- 2 puertos de salida: Data_Out y EnableOut.
- Truncamiento de trellis.

Las mejoras que hemos añadido son:

1. La especificación es que la profundidad de memoria sea fija e igual a 36 símbolos. Sin embargo, nuestro decodificador puede implementarse con cualquier profundidad de memoria, para ello solamente hay que modificar dos constantes. Aprovechando esta característica, hemos realizado el proceso completo de: implementación, simulación y verificación de 6 decodificadores diferentes. Con profundidades de memoria = 8, 24, 32, 36, 48 y 60. Comparamos todos ellos entre sí y con el Xilinx IP core Viterbi decoder v5.0. En el *tema 7* mostramos las diferencias que existen entre ellos en área ocupada y rendimiento corrector de errores.

2. Hemos realizado una arquitectura modular, en la que instanciamos bloques simples hasta conseguir la arquitectura completa. Esto nos ha permitido realizar diversas configuraciones serie-paralelo, con las que se obtienen diferentes compromisos entre mínima área ocupada y máxima frecuencia de reloj. El mejor compromiso lo obtenemos mediante la estructura 8 etapas serie * 8 etapas paralelo. Por ese motivo en el decodificador empleamos esta estructura 8*8. Lógicamente, en la estructura serie utilizamos la técnica pipeline, consultar [46], [47] y [48] del *tema 7*.

3. Al tener 8 etapas serie, se necesitan $8T_{CLKs}$ para decodificar cada dato de entrada. Por tanto, el período de proceso de un dato es $8T_{CLKs}$. Esto obliga a que la frecuencia de reloj del decodificador sea 8 veces mayor que la frecuencia con la que llegan datos a su entrada. Entonces el decodificador trabaja con F_{CLK}, en la entrada debe llegar un dato cada $8T_{CLKs}$, y en la salida se obtiene un dato cada $8T_{CLKs}$.

4. La especificación indica que la secuencia de entrada al decodificador será siempre continua. Sin embargo, hemos ampliado esta especificación, de manera que *ViterbiDecoder.vhd* acepta una secuencia de entrada continua o divida en tramas, con longitud e intervalo entre tramas variable. Como se emplea truncamiento de trellis, en las tramas de entrada no es necesario incluir bits de relleno al principio ni al final de la trama, únicamente habrá bits de datos. Los únicos requisitos son:

    - Longitud de trama ≥ profundidad de memoria + 1.
    - Espaciado entre tramas ≥ 8 T_{CLKs}.

5. La latencia es igual a: [10 + (profundidad de memoria +1)*8] T_{CLKs} = (profundidad de memoria +2,25) períodos de proceso de un dato. Este valor es muy bueno, porque el valor mínimo ideal es igual a profundidad de memoria períodos de proceso de un dato. (En un decodificador ideal, el primer dato de salida estaría disponible justo cuando se hubiese rellenado por completo la memoria del decodificador). Comparándolo con el IP Viterbi decoder v7.0 de Xilinx, nuestro decodificador tiene una latencia mucho más baja, como demuestran las siguientes fórmulas. (Las unidades son períodos de proceso de un dato):

    - Latencia ViterbiDecoder = profundidad de memoria + 2,25.
    - Latencia IP core Xilinx (modo normal) ≈ 4*(profundidad de memoria) + constraint length + tasa de salida. Página 27 de [43] del *tema 1*.
    - Latencia IP core Xilinx (latencia reducida) ≈ 2*(profundidad de memoria) + constraint length + tasa de salida. Página 28 de [43] del *tema 1*.

Al sintetizar sobre la FPGA Virtex 4 xc4vsx55 10ff1148, obtenemos para el decodificador con profundidad de memoria 36:

- 7060 slices (28%).
- 3712 slice flip flops (7%)
- 12488 LUTs de 4 entradas (25%).
- Máxima F_{CLK}=100,884 MHz.
- Velocidad máxima de decodificación =MaxF_{CLK}/8=12,615 Mega símbolos/s.
- Los resultados completos de síntesis para todas las profundidades de memoria están en la *tabla 7.4*.

10.3 Características decoderverilog.v.

El decodificador Viterbi que constituye el objetivo fundamental en este proyecto es *ViterbiDecoder.vhd*, al que denominamos decodificador UC3M. Sin embargo, en las fases iniciales de este proyecto desarrollamos otro decodificador, partiendo de un modelo de la Web de Opencores, al que denominamos *decoderverilog.v*. Tuvimos que realizar este segundo diseño porque fue necesario disponer de un decodificador en un plazo muy breve de tiempo, y era inviable finalizar el *ViterbiDecoder.vhd* en ese plazo.

Al realizar la simulación observamos que el decodificador de Opencores tiene errores funcionales, porque no implementa con exactitud el algoritmo de Viterbi. Debido a estos fallos, su rendimiento es inferior al que se obtiene con un decodificador Viterbi exacto, como por ejemplo *ViterbiDecoder.vhd*.

Tras sintetizar *decoderverilog.v*, descubrimos que no se cumplen las reglas de diseño síncrono, porque hay latches. Un último inconveniente, es que si la decodificación se divide entramas, hay que finalizar cada trama con una cadena de 36 ceros.

Utilizamos los simuladores que hemos codificado con VHDL y System Generator. Son modelos genéricos que nos permiten simular cualquier decodificador Viterbi. Por tanto son válidos para el decodificador UC3M o el de Opencores sin realizar cambios, únicamente hay que instanciar un decodificador u otro. Esta característica nos permite minimizar el tiempo necesario para simular decoderverilog.v. Esta tarea no supone coste de desarrollo en horas de ingeniería. El coste es únicamente el tiempo de computación necesario para ejecutar las simulaciones.
No nos ha sorprendido el bajo rendimiento y los problemas del decodificador de Opencores. Porque los decodificadores Viterbi disponibles no son libres, se requiere una licencia para utilizarlos. Sin embargo este módulo es gratuito, por tanto es lógico esperar que sus prestaciones sean inferiores a las de los módulos que requieren licencia.

Estos problemas no suponen ningún contratiempo para el proyecto, porque se trata de un módulo que utilizamos únicamente de manera temporal. Por este motivo, no hemos tratado de solucionar los problemas encontrados en decoderverilog.v, porque esto nos hubiese desviado de nuestro objetivo más importante, que es desarrollar un decodificador Viterbi propio. Por tanto, una última consideración importante es que el diseño de ViterbiDecoder.vhd no guarda ninguna relación con decoderverilog.v, ni con ningún otro decodificador. Se trata de un diseño realizado completamente en este proyecto.

10.4 Trabajos futuros.

El módulo final del proyecto, *ViterbiDecoder.vhd*, cumple todas las especificaciones requeridas, indicadas en el *apartado 1.4*. Incluso hemos ampliado esas especificaciones añadiendo algunas mejoras, que indicamos en el apartado anterior.

Nuestro trabajo se podría continuar en el futuro añadiendo nuevas funcionalidades al decodificador. Aprovechando que su estructura es modular, se pueden incluir características que no son necesarias para el proyecto, pero que mejorarían las prestaciones del decodificador. Las características que se pueden añadir son:

- Decodificación soft (blanda). Nuestro decodificador emplea decodificación dura (1 bit). Por tanto, un posible trabajo futuro sería añadir decodificación soft. Emplear una cuantificación de 3 bits conseguiría una ganancia de 2 dBs en las gráficas BER frente a Eb/No, respecto a la dura. Emplear cuatro bits proporcionaría un poco más de ganancia que 3 bits, pero la mejora no sería significativa.

- Cola de ceros (*zero tail*). En el *apartado 7.7.4* vimos las diferencias entre las distintas técnicas. En nuestro decodificador empleamos truncamiento de trellis, porque es la técnica que se indica en las especificaciones. Se trata de la mejor opción si la cadena de entrada es siempre continua. Porque se consigue el mismo rendimiento corrector de errores que con la cola de ceros, y no obliga a terminar las tramas de entrada con una cola de K-1 ceros. Sin embargo, en el caso de que la secuencia de entrada al decodificador se divida en tramas, se suele emplear cola de ceros. Porque así se consigue un mayor rendimiento corrector de errores.

- El resto de parámetros del decodificador: contraint length y tasa o decode rate son la base de la arquitectura. También pueden modificarse en un trabajo futuro, pero esta modificación exigiría grandes cambios en el código.

Emulación del codificador y el decodificador en la FPGA Virtex 4 xc4vsx55.

Hemos realizado una simulación exhaustiva, mediante la cual verificamos que el código que hemos implementado funciona correctamente. Por tanto, tenemos una seguridad de prácticamente el 100 %, de que el circuito se va a comportar en el hardware real exactamente igual que en la simulación. Pero no se puede garantizar con una seguridad absoluta del 100 %, para ello la única manera es realizar una emulación sobre la placa.

La emulación es la última fase del diseño hardware. Consiste en sintetizar el circuito en una placa hardware y verificar su funcionamiento sobre el hardware real

En simulación comprobamos que un circuito, denominado UUT, unit under test (equipo a testear), funciona tal y como esperamos. Para ello generamos vectores de test que cubren todas las combinaciones posibles en las señales de entrada al circuito. A continuación, se aplican esas señales al UUT y se observan sus salidas. Asegurándonos de que se cumplen todas las condiciones necesarias y que los resultados son siempre los esperados. Todo esto se realiza mediante una aplicación software en un PC.

El proceso de emulación es equivalente al de simulación, se trata de probar el funcionamiento del mismo circuito UUT. La diferencia está en que emulación el circuito se implementa realmente en hardware y las señales que se aplican en su entrada y se obtienen en su salida son reales. Los pasos que hay que seguir son:

1. Se parte del UUT testeado en simulación. En nuestro caso es un modelo del sistema FEC, que incluye: el codificador convolucional, canal y decodificador Viterbi. Lo denominamos *SimulacionCompleta.vhd*.

2. Sin realizar ningún cambio en *SimulacionCompleta.vhd* se obtiene el archivo de configuración de la FPGA: *simulacioncompleta.bit*. Este archivo es el UUT que se va a testear en emulación. Ya lo hemos obtenido en nuestro proyecto, porque se genera automáticamente mediante las herramientas de síntesis.

3. Se implementa *simulacioncompleta.bit* en una tarjeta hardware. Al ser multiplataforma se puede implementar sobre cualquier hardware. En nuestro caso se hará sobre la FPGA Virtex 4 xc4vsx55.

4. Se generan las señales externas que se aplicarán a los pines de entrada al circuito. Para ello son necesarios un oscilador con $F_{CLK} \leq 100{,}8$ MHz y generadores de señal.

5. Los pines de salida se conectan a un osciloscopio digital.

6. No es suficiente con la visualización de las señales de salida en la pantalla del osciloscopio. Porque se trata de secuencias de millones de bits, y por tanto es imposible analizarlas completamente de manera visual. Debemos obtener los valores de: BERoutDecoder, BERinDecoder y SERinDecoder. Porque son fundamentales para determinar si el decodificador Viterbi funciona correctamente. Para obtener estos valores, en primer lugar hay que almacenar las secuencias de salida del decodificador Viterbi en la memoria del osciloscopio digital. A continuación, hay que desarrollar una aplicación software que lea los datos almacenados en la memoria, y a partir de ellos obtenga los valores de BER y SER requeridos.

En las siguientes figuras se muestra la estructura de *simulación.completa.bit* y un esquema del montaje que habría que realizar en el laboratorio.

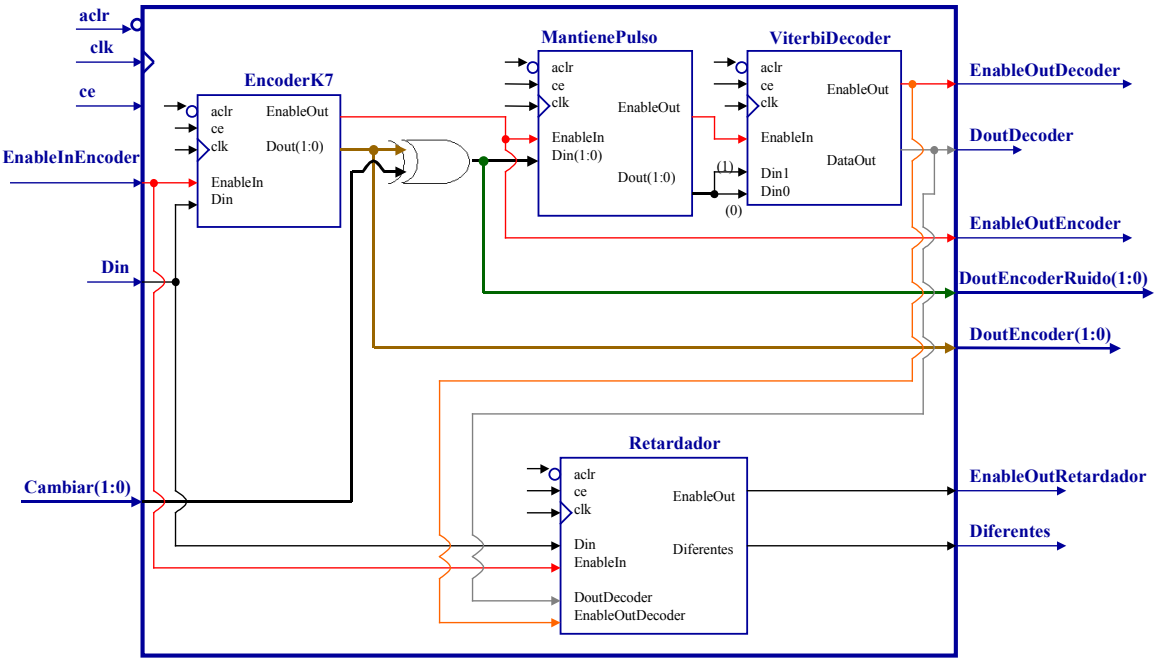


Figura 10.1: Arquitectura UUT a emular (simulacioncompleta.bit).

En el *apartado 6.5*, *figura 6.9*, describimos con todo detalle los elementos del UUT testeado en simulación. Los UUTs para emulación y simulación son exactamente el mismo circuito hardware. Por tanto, la descripción de los elementos de la *figura 6.9* se aplica a la *10.1*, así que no es necesario volver a detallarlos.

Hay que generar las señales de entrada a *simulacióncompleta.bit* y conectar sus salidas a un osciloscopio, para observar los resultados. El montaje requerido se muestra en la *figura 10.2*. Además, la comparación de las *figuras 6.9*, *10.1* y *10.2*, muestra claramente las diferencias entre simulación y emulación:

- En simulación, *figura 6.9*, desarrollamos un simulador que genera las señales de entrada que se aplican al UUT. Y el mismo simulador lee los resultados y los interpreta.

- En emulación, *figuras 10.1* y *10.2,* el UUT es el mismo. La diferencia está en que las señales de entrada se generan mediante generadores de señal y las de salida se visualizan en un osciloscopio.

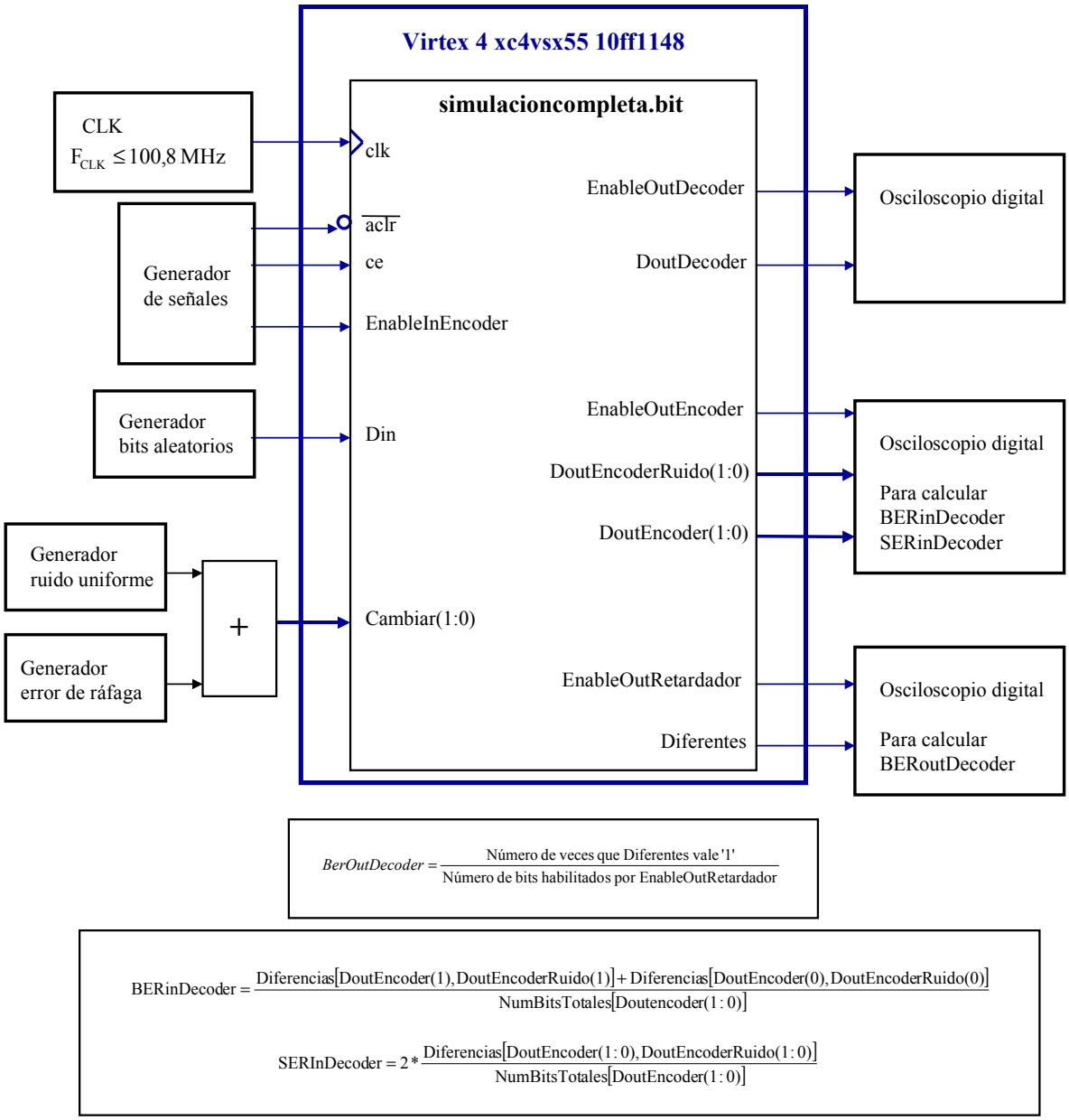


Figura 10.2: Elementos necesarios para emular ViterbiDecoder.vhd.

ANEXO A

A. LISTADO CÓDIGO FUENTE Y SIMULADORES.

A.1 Organización de los diseños en proyectos y carpetas.

El objetivo del proyecto es el de desarrollar un decodificador Viterbi con: K=7, 64 estados, R=1/2, polinomio generador 1111001, 1011011 y profundidad de memoria 36.

Además le hemos dado el valor añadido de que la profundidad de memoria es variable, porque depende únicamente de dos constantes. Aprovechando esta ventaja hemos creado un total de 6 diseños diferentes con profundidades de memoria: 8, 24, 32, 36, 48 y 60. Estos diseños sólo se diferencian en cuatro constantes, que indicamos en *A.2.4*. Pero para facilitar la clasificación y el acceso a cada uno de ellos los hemos almacenado en carpetas diferentes. El módulo principal se denomina ViterbiDecoder.vhd y tiene el mismo nombre para todos los diseños.

Además hemos desarrollado otro decodificador con profundidad de memoria 32 partiendo del modelo de la Web de Opencores. El módulo principal se denomina decoderverilog.v.

Así que en total tenemos 7 diseños diferentes que almacenamos en 6 carpetas:

1. ViterbiDecoderUC3MK7R05Depth8-171-133
2. ViterbiDecoderUC3M-K7R05Depth24-171-133
3. ViterbiDecoderUC3M-K7R05Depth36-171-133
4. ViterbiDecoderUC3M-K7R05Depth48-171-133
5. ViterbiDecoderUC3M-K7R05Depth60-171-133
6. Opencores.

Las carpetas 1 a 5 contienen un proyecto con el mismo nombre que la carpeta, realizado con la herramienta Xilinx ISE 8.1, en el que se organiza todo el código VHDL. En el nombre del proyecto se indica la profundidad de memoria del diseño.

La carpeta 6 contiene dos diseños: un ViterbiDecoder.vhd con profundidad de memoria 32 y el decoderverilog.v. Los almacenamos juntos porque hemos desarrollado un simulador que nos permite simular los dos al mismo tiempo, y con las mismas condiciones. (A los dos les llegan exactamente los mismos símbolos de entrada).

Hemos realizado todo el proceso de síntesis, implementación y simulación para cada uno de los diseños. Este es el motivo principal de tener una carpeta y proyecto diferente para cada diseño, porque podemos almacenar y clasificar los resultados de todos ellos.

También hemos implementado archivos de configuración (.bit) para la FPGA Virtex 4 xc4vsx55 10ff1148, para cada uno de los diseños. En cada una de las 6 carpetas descritas anteriormente se almacenan los archivos de configuración correspondientes a cada diseño en concreto. Además en la carpeta ArchivosDeConfguracion__bit juntamos los archivos de configuración de todos ellos.

En la carpeta WiMAX se almacena el código original del proyecto fin de carrera realizado por David Díaz Martín y Roberto Prieto Alonso. En este proyecto no hemos desarrollado este código, pero lo incluimos porque lo utilizamos para incluir nuestro módulo decodificador Viterbi.

A.2 Archivos principales.

A.2.1 Módulos codificador y decodificador.

1. **ViterbiDecoder.vhd**: Módulo principal del decodificador Viterbi que hemos implementado en este proyecto, también lo denominamos decodificador UC3M. Hay un total de 6 diseños diferentes, con profundidades de memoria: 8, 24, 32, 36, 48 y 60. Pero sólo se diferencian entre ellos en dos constantes: DecodingDepth y DecodingDepth_1, situadas en *constantes.vhd*.

2. **decoderverilog.v**: Módulo principal del decodificador Viterbi que hemos desarrollado partiendo del modelo de la Web de Opencores. Sólo hay un diseño, con profundidad de memoria 32.

3. **EncoderK7.vhd**: Codificador convolucional con estas especificaciones: Cconv(2,1,K=7), no sistemático, no recursivo, con 64 estados y polinomio generador 1111001 y 1011011. Es el mismo archivo para todos los diseños, tanto para el decodificador UC3M como el de Opencores, y para todas las profundidades de memoria.

Hemos implementado el archivo de configuración para la FPGA Virtex 4 xc4vsx55 10ff1148 para cada uno de los módulos principales:

1. **viterbidecoder.bit** (para todas las profundidades de memoria).
2. **decoderverilog.bit**
3. **encoderk7.bit.**

A.2.2 Simuladores realizados con código VHDL.

Para realizar la simulación y verificación del decodificador Viterbi y el codificador convolucional, hemos implementado dos simuladores completos con la estructura básica de un sistema de comunicaciones FEC, mostrada en las *figuras 2.1 y 6.1*. Nos permiten obtener automáticamente todos los parámetros de interés: BERoutDecoder, BERinDecoder, SERinDecoder, número total de bits decodificados y número de bits erróneos. (SER es symbol error rate).

Los dos modelos tienen la misma funcionalidad y permiten realizar simulación funcional y post place & route:

- **TestSimulacionCompleta.vhd**: Permite simular y verificar el ViterbiDecoder.vhd con cualquier profundidad de memoria, o el decoderverilog.v, pero no al mismo tiempo. Los bits de entrada se obtienen mediante un generador pseudoaleatorio incluido en la estructura del propio simulador.

- **TestLeeFichero.vhd**: Permite simular y verificar el ViterbiDecoder.vhd con cualquier profundidad de memoria, o el decoderverilog.v, pero no al mismo tiempo. Los bits de entrada se leen del fichero BitsSimulacion.txt. Esta es la única diferencia entre los dos modelos. El fichero BitsSimulacion.txt se obtiene ejecutando GenFichAleatorio.m.

Una ventaja importante es que la estructura de los dos simuladores es la misma, esto permite que ambos utilicen el mismo UUT, (unit under test). Hemos desarrollado dos UUTs con la misma estructura, sólo se diferencian en que uno instancia el ViterbiDecoder.vhd y otro el decoderverilog.v:

1) **SimulacionCompleta.vhd**: Módulo principal que incluye todos los bloques del sistema FEC necesarios para simular el ViterbiDecoder.vhd, con cualquier profundidad de memoria. La diferencia entre los módulos SimulacionCompleta.vhd con distintas profundidades de memoria está únicamente en dos constantes: DecodingDepth y DecodingDepth_1.

2) **SimulacionCompletaVerilog.vhd**: Módulo principal que incluye todos los bloques del sistema FEC necesarios para simular el decoderverilog.v.

También hemos generado los archivos de configuración de la FPGA: **simulacioncompleta.bit** y **simulacioncompletaverilog.bit**.

Entonces TestSimulacionCompleta.vhd puede instanciar SimulacionCompleta.vhd o SimulacionCompletaVerilog.vhd indistintamente, pero sólo uno de ellos a la vez. Lo mismo sucede con TestLeeFichero.vhd.

De manera que los dos modelos nos permiten verificar todos los diseños VHDL y Verilog que hemos implementado en este proyecto. Para poder realizar las simulaciones de la forma más rápida y sencilla posible, almacenamos los dos modelos y sus UUTs correspondientes en cada una de las carpetas con diseños.

La estructura de TestSimulacionCompleta.vhd almacenado en las distintas carpetas es exactamente igual en todas ellas, salvo estas diferencias mínimas:

- Para simular el ViterbiDecoder.vhd se instancia SimulacionCompleta.vhd. Para el decoderverilog.v se instancia SimulacionCompletaVerilog.vhd.

- No es posible simular los dos decodificadores al mismo tiempo. Esta opción sólo se permite en los simuladores desarrollados con System Generator.

- En SimulaciónCompleta.vhd el módulo ViterbiDecoder.vhd puede tener cualquier decoding depth. Lo único que hay que tener en cuenta para la simulación es que la constante EsperaTramasMin debe ser ≥ decoding depth + 4. Esta constante está en el código del simulador.

La estructura de TestLeeFichero.vhd almacenado en las distintas carpetas es exactamente igual en todas ellas, salvo esta diferencia:

- Para simular el ViterbiDecoder.vhd se instancia SimulacionCompleta.vhd. Para el decoderverilog.v se instancia SimulacionCompletaVerilog.vhd. No es posible simular los dos decodificadores al mismo tiempo.

- El módulo ViterbiDecoder.vhd instanciado puede tener cualquier decoding depth. En este caso no hay que modificar ninguna constante en el código del simulador.

A.2.3 Simuladores realizados con System Generator.

Hemos desarrollado simuladores con System Generator, que son los siguientes:

a) **ViterbiDecoderUC3M_Xilinx_Depth36_Simulacion_K7R05_171_133.mdl**: Este es el más completo, porque simula dos decodificadores al mismo tiempo: el que hemos desarrollado, ViterbiDecoder.vhd; y el que tomamos como referencia, IP core Viterbi decoder v5.0 de Xilinx. A los dos decodificadores les llegan exactamente los mismos símbolos de entrada. De manera que podemos compararlos al mismo tiempo con total exactitud, puesto que las condiciones de la prueba son las mismas para ambos. Este simulador es fundamental para nuestro proyecto, porque nos permite asegurarnos de que ViterbiDecoder.vhd se comporta exactamente igual que el core de Xilinx. De manera que podemos verificar que nuestro decodificador funciona correctamente, cumpliendo fielmente el algoritmo de Viterbi.

b) **ViterbiDecoderUC3M_Depth36_Simulacion_K7R05_171_133.mdl**: Simulación y síntesis de ViterbiDecoder.vhd. Este modelo trabaja con una sola frecuencia de reloj: $F=1/T_{CLK}$, de manera que el decodificador UC3M emplea $8T_{CLKs}$ en decodificar un dato.

c) **ViterbiDecoderUC3M_Depth36_8TCLK_TCLK_K7R05_171_133.mdl**: Simulación y síntesis de ViterbiDecoder.vhd. Este modelo trabaja con dos frecuencias de reloj: $F_2=1/T_2=1/(8T_{CLK})$ y $F_1=1/T_1=1/T_{CLK}$. El decodificador UC3M trabaja con la frecuencia $F_1=8F_2$. De esta manera en un sólo período T_2 es capaz de decodificar un dato. El decodificador de Xilinx trabaja con F_2.

d) **ComparaEncoderUC3M_Vs_EncoderXilinx.mdl**: Compara el codificador convolucional que hemos desarrollado en el proyecto, EncoderK7.vhd, con el que proporciona SystemGenerator, ConvolutionalEncoderv3_0 que es un core de Xilinx. El objetivo de este simulador es asegurarnos de que el Encoderk7.vhd funciona correctamente. Es decir, los bits en su salida son realmente los correspondientes a realizar una codificación convolucional 171, 133, de los bits de entrada.

e) **ViterbiDecoderUC3M_Opencores_Depth32_Simulacion_K7R05_171_133.mdl:** Este modelo simula al mismo tiempo el decoderverilog.v y el ViterbiDecoder.vhd.

f) **ViterbiDecoderUC3M_Depth36_SoloSintesis_K7R05_171_133.mdl**: Síntesis de *ViterbiDecoder.vhd*. Sólo incluye el bloque del decodificador. De esta manera se puede ver el área que ocupa el bloque en solitario. Este archivo no es un simulador.

Es suficiente con realizar la simulación de ViterbiDecoder.vhd con el modelo "a" porque es el más completo. Además el "b" y el "c" están terminados y funcionan correctamente, por lo que también pueden utilizarse para realizar la simulación. Estos modelos "b" y "c" sólo instancian el ViterbiDecoder.vhd, no incluyen el de Xilinx. Pero tienen la ventaja de son más rápidos porque sólo instancian un decodificador. Esto es de gran utilidad cuando es necesario realizar una simulación muy exhaustiva con millones de símbolos de entrada.

Hemos realizado el proceso completo de síntesis, implementación y generación del archivo de configuración .bit de los modelos "b", "c", "e" y "f" de System Generator. Por lo que disponemos de todos los archivos de configuración *.bit de la FPGA. Del modelo "a" no se puede obtener el archivo de configuración .bit porque se necesita la licencia del IP core de Xilinx.

El simulador "d" también es fundamental en nuestro proyecto. Porque para verificar que el decodificador es correcto, le hacemos decodificar una secuencia codificada convolucionalmente mediante EncoderK7.vhd. Pero antes es obligatorio verificar que el EncoderK7.vhd es un codificador convolucional funcionalmente correcto.

De nuevo, para facilitar la clasificación y el acceso a los diseños, almacenamos una copia de todos los simuladores en cada una de las carpetas con código fuente. Salvo el "e" que sólo tiene sentido utilizarlo en la carpeta Opencores.

La estructura de los simuladores almacenados en cada carpeta es la misma y sólo se diferencian en estos cambios mínimos:

- Se instancia el ViterbiDecoder.vhd con la profundidad de memoria que corresponda. O el decoderverilog.v, en el caso de: ViterbiDecoderUC3M_Opencores_Depth32_Simulacion_K7R05_171_133.mdl.

- Se instancia el IP core de Xilinx con el decoding depth correspondiente a cada diseño.

- La constante ACLR_Desactivado depende de la profundidad de memoria. Hay que poner el valor adecuado para cada caso, consultar *tabla 6.3*.

- Realizamos otro cambio en el propio nombre del archivo. A continuación de la palabra Depth indicamos la profundidad de memoria del diseño. Esto no ocurre en los simuladores realizados en VHDL, donde el nombre del archivo siempre es el mismo en todos los diseños.

- El codificador convolucional es el mismo en todos los diseños. Por eso el simulador "d" es exactamente igual en todas las carpetas.

A.2.4 Modificaciones para variar la profundidad de memoria.

Ye hemos indicado todos los cambios necesarios. Pero para dejar este aspecto lo más claro posible, juntamos todos ellos en este apartado. Se trata de las únicas modificaciones que hay que realizar en el código para adaptarlo a las distintas profundidades de memoria del ViterbiDecoder.vhd:

1. Constantes DecodingDepth y DecodingDepth_1. Situadas en constantes.vhd.

2. Constante EsperaTramasMin debe ser ≥ decoding depth + 4. Esta constante está solamente en TestSimulacionCompleta.vhd.

3. Constante ACLR_Desactivado, en los simuladores "a", "b", "c" y "e" realizados con System Generator.

A.3 Código fuente VHDL y Verilog.

A.3.1 Decodificador y codificador UC3M.

Los archivos definitivos son:

ViterbiDecoder.vhd: Código fuente del decodificador Viterbi.
viterbidecoder.bit: Archivo de configuración de la FPGA.
EncoderK7.vhd : Código fuente del codificador convolucional.
encoderk7.bit: Archivo de configuración de la FPGA.

El codificador consta únicamente de un fichero. Pero el decodificador consta de múltiples módulos internos, cuya jerarquía es la siguiente:

- ViterbiDecoder.vhd
  - ACS.vhd
    - UpdateStateInit.vhd
    - DistanceJIandHI.vhd
    - DistanceLastStateBitIn.vhd
    - normalice.vhd
      - FindMinIndex.vhd
        - Comparator4IN.vhd
  - RegisterExchange.vhd
    - calculatePaths.vhd
      - memoriaRAM.vhd
    - ExitDatoOut.vhd
      - FindMinIndex.vhd
        - Comparator4IN.vhd
  - constantes.vhd

Hemos implementado el decodificador con el valor añadido de que es inmediato cambiar su profundidad de memoria, para ello sólo hay que modificar dos constantes: DecodingDepth y DecodingDepth_1, situadas en constantes.vhd. Con esto tenemos el ViterbiDecoder.vhd configurado. Aprovechando esta ventaja hemos implementado un total de 6 decodificadores completos, con profundidades de memoria: 8, 24, 32, 36, 48 y 60.

A continuación explicamos brevemente la funcionalidad de cada módulo:

- **ACS.vhd**: Módulo Add Compare Select.

- **UpdateStateInit.vhd**: Se utiliza para tratar el caso particular que sucede en los primeros datos de la trama de entrada al decodificador.

- **DistanceJIandHI.vhd**: Calcula la distancia hasta llegar a un estado i del trellis desde sus dos estados anteriores j y h.

- **DistanceLastStateBitIn.vhd**: Calcula los parámetros necesarios para representar un estado i en el trellis.

- **normalice.vhd**: Cada NormalizeTime períodos de proceso se normalizan las 64 distancias presentes en la malla trellis, restándolas la distancia mínima que haya obtenido este módulo. Con esto se evita que crezcan indefinidamente y hace imposible que se produzca un desbordamiento (overflow).

- **FindMinIndex.vhd**: Su entrada es un vector con 64 distancias y encuentra el índice de la casilla del vector que contiene la distancia mínima.

- **Comparator4IN.vhd**: Es un módulo interno de FindMinIndex. Su entrada consiste en un vector con 64 distancias y cuatro índices del vector. El módulo busca las 4 casillas indexadas y obtiene en la salida el índice de la casilla con un valor menor.

- **RegisterExchange.vhd**: Bloque SMU, Survivor Memory Unit. Realizado con la técnica intercambio de registros.

- **calculatePaths.vhd**: Gestiona la memoria que contiene los caminos supervivientes, los actualiza en cada período de proceso y obtiene en el puerto de salida Path el primer bit de cada uno de ellos. El vector Path se envía al módulo ExitDatoOut.

- **memoriaRAM.vhd**: Es una memoria RAM de 4 filas con DecodingDepth bits cada una.

- **ExitDatoOut.vhd**: Analiza los 64 caminos supervivientes, encuentra el que tenga la distancia mínima en su última posición en t_x, y elige su primer bit como el ganador tras todo el proceso de decodificación. Las salidas de este bloque: EnableOut y DatosOut son las salidas del decodificador.

- **constantes.vhd**: Constantes empleadas por el codificador, el decodificador y los simuladores. Hay un único fichero de constantes para todo el proyecto.

A.3.2 Decodificador Opencores.

Tras descargar los archivos de la Web de Opencores y seguir los pasos que describimos en *4.5.1* se obtiene el código del decodificador. El módulo principal que se obtiene automáticamente se denomina decoder0.v.

Desarrollamos un nuevo archivo en Verilog, el **decoderverilog.v**, que se convierte en el módulo principal del decodificador. Gracias a este módulo modificamos los puertos de entrada y salida originales para ajustarlos exactamente a nuestras especificaciones.

De manera que la jerarquía de módulos es la siguiente.

- decoderverilog.v
  - decoder0.v
    - vit2.v
      - pe0.v
        - butfly2.v
          - acs2.v
          - brameter2.v
        - smu0.v
      - ctrl.v
        - delayT.v
    - trabacknew2.v
    - virtual_mem.v
    - filo.v
  - ConstantesOpencores.vhd

decoderverilog.v y ConstantesOpencores.vhd los desarrollamos completamente en este proyecto, partiendo de cero. El resto de módulos internos se obtienen automáticamente mediante los archivos descargados de la Web de Opencores. Pero sobre ellos realizamos algunos cambios en el código original para adaptarlo a nuestras especificaciones y para hacerlo compatible con System Generator. Estos cambios se describen en *4.5.2*.

A.3.3 Simuladores y estructura UUTs.

En *A.2.2* describimos los simuladores realizados con código VHDL:

- **TestSimulacionCompleta.vhd**.
- **TestLeeFichero.vhd**.

Los dos nos permiten simular el decodificador UC3M y el de Opencores. Para ello hemos desarrollado dos módulos UUT (Unit Under Test) diferentes: SimulaciónCompleta.vhd y SimulacionCompletaVerilog.vhd

SimulaciónCompleta.vhd incluye todos los módulos del sistema FEC necesarios para simular el decodificador UC3M:

- **ViterbiDecoder.vhd**: Módulo principal del decodificador Viterbi.
- **EncoderK7.vhd**: El codificador convolucional (2,1,K=7).
- **MantienePulso.vhd**: El período de proceso de un dato en el codificador es de 1 T_{CLK} y en el decodificador de 8 T_{CLKs}. Mediante este módulo adaptamos la salida del codificador a la entrada del decodificador, para poder trabajar con una única frecuencia de reloj en el simulador.
- **Retardador.vhd**: Aplica un retardo igual a la latencia del sistema a los bits de entrada al codificador, para compararlos con los de salida del decodificador. Este retardo nos permitirá calcular BEROutDecoder.
- **EscrituraFicheros.vhd**: Es un paquete que contiene las funciones necesarias para que el simulador maneje cadenas de caracteres y pueda escribir los resultados en ficheros de texto y en la consola.
- **constantes.vhd**: Constantes empleadas por el codificador, el decodificador y los simuladores. Hay un único fichero de constantes para todo el proyecto.

SimulaciónCompletaVerilog.vhd incluye todos los módulos del sistema FEC necesarios para simular el decodificador de Opencores. El único cambio entre ambos UUTs es sustituir ViterbiDecoder.vhd por decoderverilog.v. El resto de módulos instanciados son los mismos.

A modo de resumen, la jerarquía de módulos es la siguiente:

Para simular ViterbiDecoder.vhd:

- TestSimulacionCompleta.vhd
  - SimulacionCompleta.vhd
    - ViterbiDecoder.vhd
    - EncoderK7.vhd
    - MantienePulso.vhd
    - Retardador.vhd
    - EscrituraFicheros.vhd
    - constantes.vhd

- TestLeeFichero.vhd
  - SimulacionCompleta.vhd
    - ViterbiDecoder.vhd
    - EncoderK7.vhd
    - MantienePulso.vhd
    - Retardador.vhd
    - EscrituraFicheros.vhd
    - constantes.vhd

Para simular decoderverilog.v:

- TestSimulacionCompleta.vhd
  - SimulacionCompletaVerilog.vhd
    - decoderverilog.v
    - EncoderK7.vhd
    - MantienePulso.vhd
    - Retardador.vhd
    - EscrituraFicheros.vhd
    - ConstantesOpencores.vhd

- TestLeeFichero.vhd
  - SimulacionCompletaVerilog.vhd
    - decoderverilog.v
    - EncoderK7.vhd
    - MantienePulso.vhd
    - Retardador.vhd
    - EscrituraFicheros.vhd
    - ConstantesOpencores.vhd

A4. código fuente System Generator.

Para implementar los diferentes simuladores en System Generator hemos desarrollado las siguientes funciones de Matlab:

- **CalculaBERinDecoder.m**: Calcula la BER y la SER en la entrada al decodificador.
- **CalculaBEROutDecoder.m**: Calcula la BER en la salida del decodificador.
- **GeneraACLR_T1.m**: Sirve para activar y desactivar ACLR entre tramas consecutivas de entrada al simulador. Se utiliza cuando el simulador trabaja con una sola frecuencia de reloj, F = 1/TCLK.
- **GeneraACLR_T8.m**: Sirve para activar y desactivar ACLR entre tramas consecutivas de entrada al simulador. Se utiliza cuando el simulador trabaja con dos frecuencias de reloj, F1=1/TCLK y F2=1/8TCLK. ViterbiDecoder.vhd trabaja con F1 y esta función con F2.
- **ComparaEncoderK7vhd_Con_ConvolutionalEncoderv3_0.m**: Compara las salidas de dos codificadores convolucionales para asegurarnos de que son exactamente iguales. Los dos codificadores son EncoderK7.vhd y un core de Xilinx: ConvolutionalEncoderv3_0.
- **EncoderK7_config.m**: Función de configuración del módulo EncoderK7.vhd.
- **ViterbiDecoder_config.m**: Función de configuración de ViterbiDecoder.vhd.
- **decoderverilog_config.m**: Función de configuración de decoderverilog.v.
- **MantienePulso_config.m**: Función de configuración de MantienePulso.vhd.
- **Retardador_config.m**: Función de configuración de Retardador.vhd.

A.5 Ficheros de texto simuladores VHDL.

Tabla A1: Ficheros de texto empleados en los simuladores VHDL.			
Nombre	Utilidad	Entrada/ Salida	Simulador en el que se emplea
BitsSimulacion.txt	Cadena de bits aleatorios de entrada al codificador. Este fichero se genera ejecutando la función de Matlab: GenFichAleatorio.m 1 bit por línea.	Entrada	TestLeeFichero.vhd
BitsInEncoder.txt	Cadena de bits aleatorios de entrada al codificador. Este fichero lo genera el propio simulador. 1 bit por línea.	Salida	TestSimulacionCompleta.vhd
BitsOutEncoder.txt	Los bits codificados tras pasar por el codificador convolucional, y antes de añadirlos el ruido. 2 bits, un símbolo por línea.	Salida	TestSimulacionCompleta.vhd y TestLeeFichero.vhd
BitsOutEncoderRuido.txt	Los bits codificados tras añadirlos el ruido y los errores de ráfaga. Constituyen los bits de entrada al decodificador. 2 bits, un símbolo por línea.	Salida	TestSimulacionCompleta.vhd y TestLeeFichero.vhd
BitsOutDecoder.txt	Los bits de salida del decodificador. 1 bit por línea.	Salida	TestSimulacionCompleta.vhd y TestLeeFichero.vhd

- **GenFichAleatorio.m**: Al ejecutar esta función se obtiene un fichero de bits aleatorios. El formato es de un bit por línea y el número de líneas del fichero resultante es un parámetro de entrada de la función. La utilizamos para obtener automáticamente BitsSimulacion.txt.

- El resto de ficheros son de salida y los genera automáticamente el simulador VHDL.

BitsOutEncoderRuido.txt y BitsOutDecoder.txt nos resultaron muy útiles en las fases iniciales del desarrollo del proyecto. En esos momentos aún no habíamos desarrollado la aplicación que calcula automáticamente: BERin, SERin, BERout y muestra los resultados en la consola. Entonces este proceso lo hacíamos manualmente, comparando los ficheros de salida con BitsSimulacion o BitsInEncoder, utilizando la función de Matlab **ComparaFicheros.m**.

ComparaFicheros.m: Es una función que tiene como entrada dos ficheros de texto que contienen un bit o un símbolo por línea. Muestra en la consola el número de bits y símbolos diferentes, y también la BER y la SER.

Los simuladores definitivos calculan automáticamente todos los parámetros de interés y muestran los resultados en la consola. Por tanto BitsOutEncoderRuido, BitsOutDecoder y ComparaFicheros.m han perdido su utilidad inicial. A pesar de esto los mantenemos porque su generación no supone ninguna desventaja en el funcionamiento del simulador. Y al usuario le pueden resultar útiles estos archivos para otras aplicaciones.

BitsSimulacion.txt, BitsInEncoder.txt y BitsOutEncoder.txt nos permiten comprobar que EncoderK7.vhd es correcto, como veremos en el siguiente apartado.

A6. Verificación de que EncoderK7.vhd es funcionalmente correcto.

Todos los codificadores con las mismas especificaciones deben funcionar exactamente igual. De manera que ante una misma cadena de entrada deben obtener la misma cadena de salida. Por tanto, si ViterbiDecoder.vhd decodifica correctamente la secuencia codificada por EncoderK7.vhd, entonces decodifica correctamente la salida de cualquier codificador convolucional con las mismas especificaciones.

Por tanto es imprescindible verificar que EncoderK7.vhd realiza la función que hemos definido en sus especificaciones. Para ello hemos desarrollado el simulador **ComparaEncoderUC3M_Vs_EncoderXilinx.mdl**. Mediante este módulo comparamos la salida de EncoderK7.vhd, con el core de Xilinx ConvolutionalEncoderv3_0. A los dos codificadores les aplicamos los mismos bits de entrada y comprobamos que ambos obtienen exactamente la misma cadena de salida. Por tanto podemos asegurar que EncoderK7.vhd se comporta igual que el core de Xilinx. Como tenemos la certeza absoluta de que el core de Xilinx es correcto, entonces podemos asegurar con un 100 % de seguridad que nuestro codificador es funcionalmente correcto.

En las fases iniciales del desarrollo del proyecto no disponíamos del simulador definitivo anterior. Por eso, para comprobar que el EncoderK7.vhd es correcto desarrollamos una función auxiliar: **ConvolutionalEncoder.m**.

ConvolutionalEncoder.m: Es un codificador convolucional con estas especificaciones: Cconv(2,1,K=7), no sistemático, no recursivo, con 64 estados polinomio generador 1111001 y 1011011. Lee los bits de entrada de un fichero, los codifica convolucionalmente y escribe los bits codificados en otro fichero.

En esta fase inicial del proyecto utilizábamos ConvolutionalEncoder.m para codificar los bits contenidos en BitsSimulacion.txt o BitsInEncoder.txt. Después comparábamos, mediante ComparaFicheros.m, el fichero de salida resultante con BitsOutEncoder.txt. Ambos ficheros deben ser exactamente iguales, y con eso podemos asegurar que EncoderK7.vhd es funcionalmente idéntico a ConvolutionalEncoder.m

Una vez desarrollado ComparaEncoderUC3M_Vs_EncoderXilinx.mdl; tanto ConvolutionalEncoder.m como ComparaFicheros.m, BitsInEncoder.txt y BitsOutEncoder.txt pierden su utilidad inicial. Sin embargo los mantenemos porque su generación no supone ninguna desventaja en el funcionamiento del simulador. Y al usuario le puede resultar útil disponer de estas funciones y archivos para otras aplicaciones.

Nota A1: Que las cadenas de salida de EncoderK7.vhd y ConvolutionalEncoder.m sean iguales, no es condición suficiente para asegurar que EncoderK7.vhd es funcionalmente correcto. Porque ambos codificadores los hemos desarrollado nosotros, así que podría darse el caso de que en los dos hubiésemos cometido el mismo error. Esto haría que las cadenas de salida de ambos codificadores fuesen iguales, pero erróneas. La comparación con el core de Xilinx sí nos permite asegurar que EncoderK7.vhd es correcto. Y una vez que tenemos la certeza absoluta de que EncoderK7.vhd es correcto, eso implica que ConvolutionalEncoder.m también lo es.

A.7 Test bench para el cálculo de estados anteriores.

En el código del decodificador necesitamos una constante Matrix64States, almacenada en constantes.vhd, que contiene:

1) Los estados j y h anteriores a cada estado i.

2) El bit de entrada al codificador m[n] que permite pasar del estado j ó h al i.

3) La salida $Dout_1$, correspondiente a aplicar la entrada m[n] al estado j ó h, con el polinomio 171.

4) La salida $Dout_0$, correspondiente a aplicar la entrada m[n] al estado j ó h, con el polinomio 133.

Una vez obtenidos los valores, se almacenan en la constante y ya no hay que modificarla más. De manera que estos valores sólo deben calcularse una vez, antes de iniciar el desarrollo del código del decodificador.

Es fundamental no cometer ni un sólo error en el cálculo de estos valores, porque un sólo fallo implicaría que el decodificador no funcionase correctamente. Además sería un fallo difícil de detectar, porque se produciría esporádicamente.

Por eso hemos desarrollado un test bench con código VHDL y unas funciones de Matlab que generan automáticamente todos los datos que necesitamos. Y también verifican automáticamente que todos los datos de la constante Matrix64States son correctos.

Las funciones y código necesarias son:

- **TBGeneraMatrizEstadoAnterior.vhd**: Se trata de un test bench que genera automáticamente los valores que deben escribirse en la constante Matrix64States. Estos valores se escriben automáticamente en **MatrizEstadoAnterior.txt**, y a continuación debemos escribirlos a mano en la constante.

- **GeneramatrizEstadoanterior.vhd**: UUT sobre el que se aplica el test bench.

- **CompruebaMatrizEstadoAnterior.m**: El objetivo de esta función es asegurarse de que los valores de la constante Matrix64States, dentro del archivo constantes.vhd, son correctos.

- **EncuentraEstadosAnteriores.m**: Esta función obtiene los dos estados anteriores, j y h, de cada estado i de un decodificador Viterbi, con código convolucional (octal) 171, 133. Los resultados se escriben en **EstadosAnterioresDecimal.txt**. Esta función no se utiliza en el código final del decodificador Viterbi, ni del codificador, ni de ninguno de los simuladores. Únicamente es útil como paso previo al diseño del código. Necesitamos conocer cuáles son los estados anteriores a cada i para poder diseñar el decodificador. Por tanto, antes de desarrollar el código ejecutamos esta función, obtenemos "EstadosAnterioresDecimal.txt" y ya no es necesario volver a ejecutar la función.

En los comentarios de los módulos y funciones se indica el proceso a seguir, detallando todos los pasos necesarios. A modo de resumen, el proceso es el siguiente:

1) Se ejecuta TBGeneraMatrizEstadoAnterior.vhd. Al hacer esto se calculan los valores de la constante y se escriben automáticamente en "MatrizEstadoAnterior.txt".

2) Rellenar a mano las filas y columnas de la constante Matrix64States, utilizando los datos de "MatrizEstadoAnterior.txt".

3) Volver a ejecutar TBGeneraMatrizEstadoAnterior.vhd. Al hacer esto, se leen automáticamente los valores que haya escritos en Matrix64States, y se escriben en el fichero MatrizEstadoAnterior.txt.

4) Ejecutar CompruebaMatrizEstadoAnterior.m para asegurarnos de que los valores escritos en MatrizEstadoAnterior.txt son correctos.

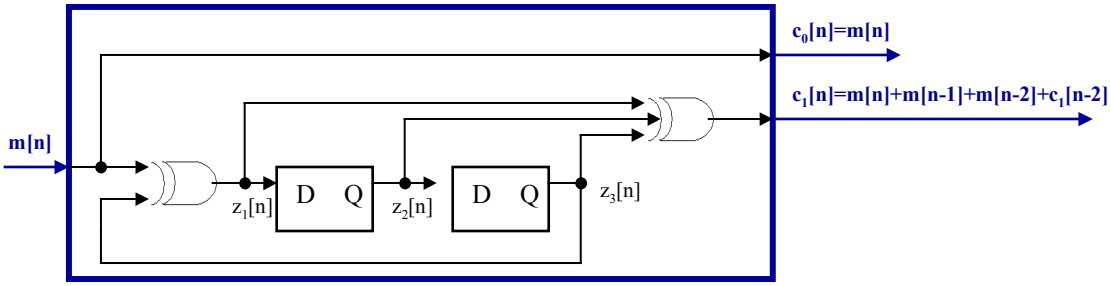
5) Ejecutar EncuentraEstadosAnteriores.m para obtener los estados j y h anteriores a cada estado i. Estos datos se almacenan automáticamente en EstadosAnterioresDecimal.txt y nos resultarán muy útiles en el diseño del decodificador.

Volvemos a insistir en que todos estos pasos sólo deben realizarse una vez, al inicio del desarrollo del decodificador. Una vez que hayamos verificado que los valores escritos en la constante Matrix64States son correctos, no será necesario volver a utilizar estas funciones, ni test benchs.

ANEXO B

B. DESARROLLO MATEMÁTICO CODIFICADOR RECURSIVO.

B1 Codificador convolucional(2, 1, m=2), sistemático recursivo.



Representación dependiente del tiempo discreto.

$$\left.\begin{aligned} z_1[n] &= m[n] + z_3[n] \\ z_3[n] &= z_1[n-2] \end{aligned}\right\} \Rightarrow z_1[n] = m[n] + z_1[n-2]$$

$$\left.\begin{aligned} z_2[n] &= z_1[n-1] = m[n-1] + z_1[n-3] \\ z_2[n] &= z_1[n-1] \;\Rightarrow z_2[n-2] = z_1[n-3] \end{aligned}\right\} \Rightarrow z_2[n] = m[n-1] + z_2[n-2]$$

$$\left.\begin{aligned} z_3[n] &= z_1[n-2] = m[n-2] + z_1[n-4] \\ z_3[n] &= z_1[n-2] \;\Rightarrow z_3[n-2] = z_1[n-4] \end{aligned}\right\} \Rightarrow z_3[n] = m[n-2] + z_3[n-2]$$

$$c_1[n] = z_1[n] + z_2[n] + z_3[n] = m[n] + m[n-1] + m[n-2] + z_1[n-2] + z_2[n-2] + z_3[n-2]$$
$$c_1[n-2] = z_1[n-2] + z_2[n-2] + z_3[n-2]$$

$$\boxed{c_1[n] = m[n] + m[n-1] + m[n-2] + c_1[n-2]}$$

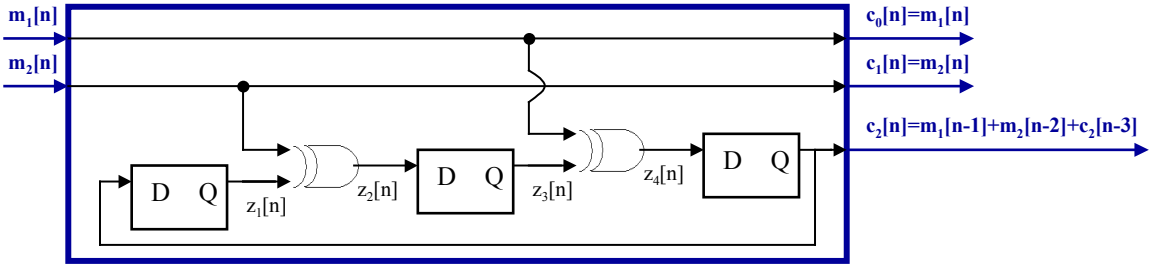
Representación en forma polinómica.

Se obtienen fácilmente a partir de las ecuaciones dependientes del tiempo discreto

$$z_1(x) = \frac{1}{1+x^2}; \qquad z_2(x) = \frac{x}{1+x^2}; \qquad z_3(x) = \frac{x^2}{1+x^2}$$

$$\boxed{c_1(x) = z_1(x) + z_2(x) + z_3(x) = m(x)\frac{1+x+x^2}{1+x^2}}$$

B2 Codificador convolucional(3, 2, m=3), sistemático recursivo.



Representación dependiente del tiempo discreto.

$$\left.\begin{aligned} z_2[n] &= m_2[n] + z_1[n] \\ z_1[n] &= c_2[n-1] \end{aligned}\right\} \Rightarrow z_2[n] = m_2[n] + c_2[n-1]$$

$$z_3[n] = z_2[n-1] = m_2[n-1] + c_2[n-2]$$

$$z_4[n] = m_1[n] + z_3[n] = m_1[n] + m_2[n-1] + c_2[n-2]$$

$$\boxed{c_2[n] = z_4[n-1] = m_1[n-1] + m_2[n-2] + c_2[n-3]}$$

Representación en forma polinómica.

Se obtienen fácilmente a partir de las ecuaciones dependientes del tiempo discreto

$$\boxed{c_2(x) = m_1(x)\frac{x}{1+x^3} + m_2(x)\frac{x^2}{1+x^3}}$$

ANEXO C

C. FUNDAMENTOS MATEMÁTICOS DECODIFICADOR VITERBI

El objetivo en este punto es demostrar que el decodificador Viterbi es el algoritmo de máxima verosimilitud, el más óptimo para decodificar un código convolucional. Pero no pretendemos detallar el funcionamiento, porque esa tarea la realizamos de manera más clara en el *apartado 2.6*.

Todo lo expuesto en este apartado es un resumen de la bibliografía, punto *2.8.1B*.

A continuación se describen las señales de interés para el sistema, ver *figura 2.10*.

En cada instante de tiempo t=n entran k bits al codificador: $m_{k-1}[n]$, ...,$m_1[n]$,$m_0[n]$.
El conjunto de k bits se denomina símbolo o dato de entrada: $x[n]=m[n]=(m_{k-1},....m_0)$.
La secuencia de entrada al codificador consta de N símbolos, el primero es: $x_1=(m_{k-1,1},...,m_{1,1},m_{0,1})$ y el último $x_N=(m_{k-1,N},...,m_{1,N},m_{0,N})$. De manera que un símbolo i de entrada al codificador es x_i y la trama completa es: $x=\{x_1,x_2...x_N\}$.

En la salida del codificador se obtiene un dato de n bits en cada instante de tiempo t=n, así que un símbolo de salida del codificador es: $c[t]=(c_{n-1}[t], ...c_0[t])$.
El conjunto de n bits lo denominamos: $DoutC[t]= c[t]=(c_{n-1}[t], ...c_0[t])$. La secuencia codificada consta de N datos de n bits: $DoutC=\{DoutC_1,DoutC_2.....,DoutC_N\}$. Donde el dato i corresponde a $DoutC_i=(c_{n-1,i},..., c_{1,i},c_{0,i})$.

DoutC[t] pasa por los bloques entrelazador, modulador y transmisor. A continuación, en cada instante de tiempo t=n se transmite un símbolo de n bits: $a[t]=(a_{n-1}[t],...,a_1[t],a_0[t])$.

La secuencia a[t] atraviesa un canal ruidoso, y en la entrada del decodificador se reciben datos de n bits: $r[t]=(r_{n-1}[t],..., r_1[t],r_0[t])$. El conjunto de n bits lo denominamos: $DinV[t]=r[t]= (r_{n-1}[t],..., r_1[t],r_0[t])$. La secuencia codificada consta de N símbolos de n bits: $DinV=\{DinV_1,...,DinV_N\}$. Donde el dato i corresponde a $DinV_i=(r_{n-11,i},...,r_{1,i},...,r_{0,i})$.

Para simplificar los cálculos nos basamos en un sistema en el que el ruido sea blanco y gaussiano, canal AWGN, additive white gaussian noise. Además el canal es BSC, binary symmetric channel. Esto significa que por el canal únicamente se transmiten bits, y la probabilidad de transmitir un '1' es la misma que la de transmitir un '0'. Además, por simplicidad consideramos que el bloque modulador emplea modulación BPSC. Con estas condiciones, la secuencia modulada a[t] es igual a la codificada c[t].

$r_i[t]= c_i[t]+n_i[t]$, donde $n_i[t]$ *es* $N(0,\sigma^2)$ y $\sigma^2 = \frac{N_0}{2}$. *Nota A2*: i es el orden del bit dentro del símbolo: $i \in [0..n-1]$.

Se asume que $n_i[t]$ es independiente tanto de i como de t, con lo que se trata de un canal sin memoria.

La función de verosimilitud es: $f(DinV[t] \mid DoutC[t]) = \prod_{i=1}^{n} p_c^{[r_i[t] \neq c_i[t]]} (1-p_c)^{r_i[t]=c_i[t]} \Rightarrow$

$$p_c^{d_H(DinV,DoutC)} (1-p_c)^{n-d_H(DinV,DoutC)} = \left(\frac{p_c}{1-p_c}\right)^{d_H(DinV,DoutC)} (1-p_c)^n$$

Donde d_H es la distancia de Hamming:

$$d_H(x,y) = \sum_{i=1}^{n} [x_i \neq y_i]; \qquad \text{siendo } [x_i \neq y_i] = \begin{cases} 1 & \text{si } x_i \neq y_i \\ 0 & \text{si } x_i = y_i \end{cases}$$

El decodificador Viterbi es el que maximiza la función de verosimilitud, lo cual es equivalente a minimizar el negativo del logaritmo de la verosimilitud. Se trabaja con logaritmos por simplicidad, para convertir los productos en sumas:

$$-\log(f(DinV[t] \mid DoutC[t])) = -d_H((DinV[t] \mid DoutC[t]))\log\left(\frac{p_c}{1-p_c}\right) - n\log(1-p_c)$$

Se utiliza $d_H(DinV[t]|DoutC[t])$ como el negativo del logaritmo de la verosimilitud, porque $\log(p_c/(1-p_c)) < 0$. De manera que maximizar la verosimilitud equivale a minimizar $d_H(DinV[t]|DoutC[t])$.

El algoritmo Viterbi tiene como entrada la secuencia: $DinV=\{DinV_1,.....,DinV_N\}$, y con ella estima cuál fue la secuencia transmitida: $DoutC=\{DoutC_1,.....,DoutC_N\}$. Y a partir de ella estima cuál fue la secuencia original de entrada al codificador: $x=\{x_1,x_2...x_N\}$.

La idea básica del decodificador Viterbi consiste en que una secuencia codificada $DoutC=\{DoutC_1,.....,DoutC_N\}$, corresponde a un camino exacto a través del trellis. Un camino exacto es el que tiene una distancia de Hamming igual a cero. Debido al ruido presente en el canal, la secuencia recibida $DinV=\{DinV_1,.....,DinV_N\}$, puede no corresponder exactamente con un camino a través del trellis. De manera que su distancia de Hamming será mayor que cero.

Entonces la función del decodificador consiste en encontrar el camino a través del trellis, que sea lo más próximo posible a la secuencia recibida DinV. Este camino es el de máxima verosimilitud, y es igual al camino a través del trellis con menor distancia de Hamming con DinV.

Para una entrada x[t], la salida DoutC[t] depende del estado del codificador S[t], que a su vez depende de las entradas previas. Esta dependencia con las entradas anteriores, implica que la decisión óptima no se puede tomar basándose sólo en la función de verosimilitud en un instante de tiempo: f(DinV[t]|x[t]). La decisión óptima debe basarse en una secuencia entera de símbolos recibidos. Entonces, la función de verosimilitud que hay que maximizar es:

$f(DinV \mid x) = f(DinV_1^N \mid x_1^N)$; Donde x_1^N es una secuencia de N símbolos $x=\{x_1,x_2...x_N\}$. Por tanto la ecuación a maximizar es:

$$f(DinV \mid x) = f(DinV_1, DinV_2...DinV_N \mid x_1, x_2...x_N) = \prod_{t=1}^{N} f(DinV[t] \mid x[t])$$

Es conveniente trabajar con logaritmos:

$$\log f(DinV \mid x) = \sum_{t=1}^{N} \log f(DinV[t] \mid x[t])$$

A continuación consideramos una secuencia $\hat{x}_1^z = \{\hat{x}_1, \hat{x}_{2,...,} \hat{x}_z\}$, que sale del codificador en el estado S[t_z]=j. Esta secuencia constituye un camino, o secuencia de estados a través del trellis. La denominamos $\Pi[t_z]=\{S_1[t_z],S_2[t_z],....,S_z[t_z]\}$.

La función logaritmo de la verosimilitud para esta secuencia es:

$$\log f(DinV_1^z \mid \hat{x}_1^z) = \sum_{i=1}^{z} \log f(DinV_i \mid \hat{x}_i)$$

$Distance_j[t_z] = -\log f(DinV_1^z \mid \hat{x}_1^z)$. A este término lo denominamos la distancia hasta el estado j en el instante t_z. Constituye la distancia, el peso, o path metric del camino seguido por la secuencia $\hat{x}_1^z$ a través del trellis y que termina en el estado j. Es la distancia desde el origen hasta el estado j en t_z.

La secuencia $\hat{x}_1^{z+1} = \{\hat{x}_1, \hat{x}_2, ..., \hat{x}_{z+1}\}$ puede obtenerse añadiendo la entrada $\hat{x}_z^{z+1}$ a la secuencia $\hat{x}_1^z$, y suponiendo que la entrada $\hat{x}_{z+1}$ lleva al estado i en t_{z+1}, S[t_{z+1}]=i. Entonces la distancia hasta el estado i en t_{z+1} es:

$Distance_i[t_{z+1}] = -\sum_{i=1}^{z+1} \log f(DinV_i \mid \hat{x}_i) = -\sum_{i=1}^{z} \log f(DinV_i \mid \hat{x}_i) - \log f(DinV[t_{z+1}] \mid \hat{x}[t_{z+1}]) \Rightarrow$

$Distance_i[t_{z+1}] = Distance_j[t_z] - \log f(DinV[t_{z+1}] \mid \hat{x}[t_{z+1}])$

$Peso_{ji}[t_{z \to z+1}] = -\log f(DinV[t_{z+1}] \mid \hat{x}[t_{z+1}])$, es el peso de la rama que pasa del estado j en t_z al i en t_{z+1}. El término original es branch metric.

El objetivo del decodificador es maximizar la verosimilitud de la secuencia DinV a través del trellis. Esto es equivalente a minimizar el negativo del logaritmo de la verosimilitud. Y esto a su vez es equivalente a minimizar la distancia de Hamming de la secuencia DinV a través del trellis.

La función del decodificador consiste en estimar los diferentes caminos de la secuencia DinV a través del trellis. A continuación calculará las distancias de Hamming de cada uno de los caminos, hasta cada uno de los estados del trellis en un instante de tiempo t_z. De todos los caminos, seleccionará el que tenga menor distancia, path metric, porque es el que maximiza la función de verosimilitud. Por último, a partir de este camino seleccionado, estimará la secuencia más próxima a la original de entrada al codificador: $x=\{x_1,x_2...x_n\}$.

El proceso es continuo y se repite para todo t_z, con $z \in [0..(N+ \text{decoding depth})]$.

ANEXO D

D. SÍNTESIS DE TODOS LOS MÓDULOS VHDL

D.1 Objetivos.

En este anexo recopilamos los resultados tras realizar la síntesis y el map de todos los módulos VHDL que hemos implementado en el proyecto. Tanto de los principales, de los que se obtiene el archivo de configuración (*.bit) de la FPGA, como de los internos. En este apartado no describiremos la arquitectura ni comentaremos los resultados. Todo esto ya lo hemos hecho en los capítulos anteriores de la memoria. Por tanto el objetivo de este apartado es solamente recopilar la información de síntesis y map, así facilitamos al máximo al lector la tarea de búsqueda de la información.

Síntesis: Es el proceso que traslada el código VHDL a los elementos más básicos que constituyen el hardware de la tecnología empleada. En el caso de una FPGA consisten en los elementos lógicos de la FPGA, flip-flops, LUTs y memoria RAM. La síntesis genera una netlist, donde se especifica la arquitectura del circuito a nivel hardware usando estos elementos básicos.

Map: Convierte los elementos lógicos básicos a CLBs e IOBs. Optimiza los recursos para aprovechar lo mejor posible los CLBs e IOBs disponibles de una FPGA en concreto. Esta optimización permite una reducción de área del diseño, frente al área que se obtiene tras la síntesis.

La información tras síntesis y map de los módulos principales es fundamental. Porque a partir de ella se obtiene el área ocupada y la frecuencia máxima de trabajo de los diseños hardware realizados.

En el desarrollo del proyecto hemos optimizado todos los módulos tanto en área como en velocidad. En este anexo mostramos los resultados tras la síntesis y map de todos los bloques internos. Esta información es importante para el lector si desea conocer detalladamente la arquitectura interna de los bloques hardware y el camino crítico. También es fundamental si en un trabajo futuro se decide continuar optimizando alguno de los módulos realizados. Aumentando su frecuencia máxima de trabajo o disminuyendo el área ocupada.

En los capítulos anteriores de esta memoria ya hemos incluido los resultados de síntesis y map de los bloques principales, esta información puede consultarse en:

1. EncoderK7.vhd → En el *tema 3* de esta memoria.
2. ViterbiDecoder.vhd → En el *tema 7*.
3. decoderverilog.v → En el *tema 7*.
4. SimulacionCompleta.vhd → en el *tema 7*.

En este anexo recopilamos la información de los 4 diseños anteriores y también de sus módulos internos.

En todos los casos, tanto en los temas anteriores de esta memoria como en este anexo, **los resultados se refieren a la FPGA Virtex 4 xc4vsx55 -10ff1148.**

También hemos realizado simuladores con System Generator que pueden sintetizarse en hardware. Hemos generado los archivos de configuración (*.bit) de todos ellos. Pero no es necesario incluir en este anexo sus resultados tras realizar la síntesis y el map. Porque su estructura básica se consigue importando cajas negras de VHDL, de manera que sus resultados serán muy parecidos a los que se obtienen con los bloques VHDL de los que proceden.

D2 Síntesis codificador convolucional EncoderK7.vhd.

Está constituido por un único bloque. Las *tablas D1* y *D2* son una copia de las *tablas 3.2* y *3.3* en el *capítulo 3*.

Tabla D1: Síntesis EncoderK7.vhd			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	6	24576	0%
Number of Slice Flip Flops	8	49152	0%
Number of 4 input LUTs	5	49152	0%
Number of bonded IOBs	8	640	1%
IOB Flip Flops	1		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	2,099 ns		476,14 MHz
Minimum input arrival time before clock	3,621 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D2: Map EncoderK7.vhd			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	5	49152	1%
Number of 4 input LUTs	5	49152	1%
Number of occupied slices	6	24576	1%
Number of Slices containing only related logic	6	6	100%
Number of Slices containing unrelated logic	0	6	0%
Total Number 4 input LUTs	5	49152	1%
Number of bonded IOBs	8	640	1%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	102		
Additional JTAG gate count for IOBs	384		
Peak memory usage	257 MB		

D3 Síntesis ViterbiDecoder.vhd con profundidad de memoria 36.

Está constituido por los bloques que listamos a continuación. Detrás del nombre de cada bloque añadimos entre paréntesis el número de veces que se instancia ese bloque en el módulo ViterbiDecoder.vhd. Esta información es útil, porque si en el futuro se decide continuar con la optimización decodificador, la mejor opción es comenzar con los bloques que se instancian más veces en el código:

- ViterbiDecoder.vhd (1)
  - ACS.vhd (1)
    - UpdateStateInit.vhd (1)
    - DistanceJIandHI.vhd (10)
    - DistanceLastStateBitIn.vhd (10)
    - normalice.vhd (1)
      - FindMinIndex.vhd (2)
        - Comparator4IN.vhd (42)
  - RegisterExchange.vhd (1)
    - calculatePaths.vhd (1)
      - memoriaRAM.vhd (32)
    - ExitDatoOut.vhd (1)
      - FindMinIndex.vhd (2)
        - Comparator4IN.vhd (42)
  - constantes.vhd

Nota D1: FindMinIndex se instancia 2 veces en todo el decodificador, no 2*2 veces. Lo mismo sucede con Comparator4IN, que se instancia un total de 42 veces.

En el *capítulo 7* describimos los aspectos más importantes de la síntesis y el map del decodificador:

- Indicamos que el camino crítico está formado por el bloque FindMinIndex. Este bloque fija la frecuencia máxima de reloj en 100,884 MHz; T_{CLK}=9,912 ns.

- En las *tablas 7.1 y 7.2* mostramos los resultados tras síntesis y map de ViterbiDecoder.vhd, para una profundidad de memoria igual a 36. Las *tablas D3* y *D4* son una copia de estas *tablas 7.1 y 7.2*.

- En la *tabla 7.4* mostramos los resultados tras sintetizar ViterbiDecoder,vhd con distintas profundidades de memoria.

Para no hacer demasiado extenso este anexo, sólo mostramos los resultados referidos a un decodificador con profundidad de memoria igual a 36.

Tabla D3: Síntesis ViterbiDecoder.vhd, profundidad de memoria=36.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	7060	24576	28%
Number of Slice Flip Flops	3712	49152	7%
Number of 4 input LUTs	12488	49152	25%
Number of bonded IOBs	8	640	1%
Number of FIFO16/RAMB16s	32	320	10%
Number used as RAMB16s	32		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	9,912 ns		100,884 MHz
Minimum input arrival time before clock	8,852 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D4: Map ViterbiDecoder.vhd, profundidad de memoria=36.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	3707	49152	7%
Number of 4 input LUTs	12047	49152	24%
Number of occupied slices	8383	24576	34%
Number of Slices containing only related logic	8383	8383	100%
Number of Slices containing unrelated logic	0	8383	0%
Total Number 4 input LUTs	12098	49152	24%
Number used as logic	12047		
Number used as rote-thru	51		
Number of bonded IOBs	8	640	1%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Number of FIFO16/RAMB16s	32	320	10%
Number used as FIFO16s	0		
Number used as RAMB16s	32		
Total equivalent gate count for design	122714		
Additional JTAG gate count for IOBs	384		
Peak memory usage	378 MB		

Tabla D5: Síntesis ACS.vhd, profundidad de memoria=36.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	4179	24576	17%
Number of Slice Flip Flops	2208	49152	4%
Number of 4 input LUTs	6450	49152	13%
Number of bonded IOBs	647*	640	101%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	9,912 ns		100,884 MHz
Minimum input arrival time before clock	9,954 ns		
Maximum output required time after clock	5,469 ns		
Maximum combinational path delay	No path found		

Tabla D6: Map ACS.vhd, profundidad de memoria=36.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	2079	49152	4%
Number of 4 input LUTs	6037	49152	12%
Number of occupied slices	4557	24576	18%
Number of Slices containing only related logic	4557	4557	100%
Number of Slices containing unrelated logic	0	4557	0%
Total Number 4 input LUTs	6070	49152	12%
Number used as logic	6037		
Number used as rote-thru	33		
Number of bonded IOBs	647	640	101%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	68571		
Additional JTAG gate count for IOBs	31056		
Peak memory usage	323 MB		

Tabla D7: Síntesis RegisterExchange.vhd, profundidad de memoria=36.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	3368	24576	13%
Number of Slice Flip Flops	1774	49152	3%
Number of 4 input LUTs	6280	49152	12%
Number of bonded IOBs	646*	640	101%
Number of FIFO16/RAMB16s	32	320	10%
Number used as RAMB16s	32		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	7,211 ns		138,682 MHz
Minimum input arrival time before clock	6,436ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D8: Map RegisterExchange.vhd, profundidad de memoria=36.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	1132	49152	2%
Number of 4 input LUTs	6252	49152	12%
Number of occupied slices	3561	24576	14%
Number of Slices containing only related logic	3561	3561	100%
Number of Slices containing unrelated logic	0	3561	0%
Total Number 4 input LUTs	6270	49152	12%
Number used as logic	6252		
Number used as rote-thru	18		
Number of bonded IOBs	646*	640	101%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Number of FIFO16/RAMB16s	32	320	10%
Number used as FIFO16s	0		
Number used as RAMB16s	32		
Total equivalent gate count for design	58031		
Additional JTAG gate count for IOBs	31008		
Peak memory usage	315 MB		

Tabla D9: Síntesis calculatePaths.vhd.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	2454	24576	9%
Number of Slice Flip Flops	890	49152	1%
Number of 4 input LUTs	4765	49152	9%
Number of bonded IOBs	261	640	40%
Number of FIFO16/RAMB16s	32	320	10%
Number used as RAMB16s	32		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	5,898 ns		169,61 MHz
Minimum input arrival time before clock	9,864 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D10: Map calculatePaths.vhd.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	826	49152	1%
Number of 4 input LUTs	4756	49152	9%
Number of occupied slices	2656	24576	10%
Number of Slices containing only related logic	2656	2656	100%
Number of Slices containing unrelated logic	0	2656	0%
Total Number 4 input LUTs	4765	49152	9%
Number of bonded IOBs	261	640	40%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Number of FIFO16/RAMB16s	32	320	10%
Number used as FIFO16s	0		
Number used as RAMB16s	32		
Total equivalent gate count for design	39043		
Additional JTAG gate count for IOBs	12528		
Peak memory usage	294 MB		

Tabla D11: Síntesis ExitDatoOut.vhd.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	1140	24576	4%
Number of Slice Flip Flops	266	49152	4%
Number of 4 input LUTs	2050	49152	13%
Number of bonded IOBs	519	640	81%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	7,211 ns		138,682 MHz
Minimum input arrival time before clock	8,115 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D12: Map ExitDatoOut.vhd.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	262	49152	1%
Number of 4 input LUTs	2022	49152	4%
Number of occupied slices	1209	24576	4%
Number of Slices containing only related logic	1209	1209	100%
Number of Slices containing unrelated logic	0	1209	0%
Total Number 4 input LUTs	2038	49152	4%
Number used as logic	2022		
Number used as rote-thru	16		
Number of bonded IOBs	519	640	81%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	19279		
Additional JTAG gate count for IOBs	24912		
Peak memory usage	276 MB		

Tabla D13: Síntesis memoriaRAM.vhd El bloque se instancia 32 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of bonded IOBs	115	640	17%
Number of FIFO16/RAMB16s	1	320	0%
Number used as RAMB16s	1		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	No patth found		
Minimum input arrival time before clock	2,262 ns		
Maximum output required time after clock	6,591 ns		
Maximum combinational path delay	No path found		

Tabla D14: Map memoriaRAM.vhd El bloque se instancia 32 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of occupied slices	0	24576	0%
Number of Slices containing only related logic	0	0	0%
Number of Slices containing unrelated logic	0	0	0%
Number of bonded IOBs	115	640	17%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Number of FIFO16/RAMB16s0	1	320	1%
Number used as FIFO16s	0		
Number used as RAMB16s	1		
Total equivalent gate count for design	68571		
Additional JTAG gate count for IOBs	31056		
Peak memory usage	323 MB		

Tabla D15: Síntesis FindMinIndex.vhd El bloque se instancia 2 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	1038	24576	4%
Number of Slice Flip Flops	198	49152	0%
Number of 4 input LUTs	1835	49152	3%
Number of bonded IOBs	458	640	71%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	7,211 ns		138,682 MHz
Minimum input arrival time before clock	8,115 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D16: Map FindMinIndex.vhd El bloque se instancia 2 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	192	49152	1%
Number of 4 input LUTs	1835	49152	3%
Number of occupied slices	1104	24576	4%
Number of Slices containing only related logic	1104	1104	100%
Number of Slices containing unrelated logic	0	11104	0%
Total Number 4 input LUTs	1835	49152	3%
Number of bonded IOBs	458	640	71%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	17190		
Additional JTAG gate count for IOBs	21982		
Peak memory usage	272 MB		

Tabla D17: Síntesis Comparator4IN.vhd Se instancia 42 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	498	24576	2%
Number of 4 input LUTs	984	49152	2%
Number of bonded IOBs	478	640	74%
Number of GCLKs	0	32	0%
Timing Summary			
Minimum period / Maximum Frequency	No path found		
Minimum input arrival time before clock	No path found		
Maximum output required time after clock	No path found		
Maximum combinational path delay	15,172 ns		

Tabla D18: Map Comparator4IN.vhd Se instancia 42 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of 4 input LUTs	984	49152	2%
Number of occupied slices	494	24576	2%
Number of Slices containing only related logic	494	494	100%
Number of Slices containing unrelated logic	0	494	0%
Total Number 4 input LUTs	984	49152	2%
Number of bonded IOBs	478	640	74%
Total equivalent gate count for design	8550		
Additional JTAG gate count for IOBs	22944		
Peak memory usage	268 MB		

Tabla D19: Síntesis UpdateStateInit.vhd			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	37	24576	0%
Number of 4 input LUTs	64	49152	0%
Number of bonded IOBs	133	640	20%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	No path found		
Minimum input arrival time before clock	3,099 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D20: Map UpdateStateInit.vhd			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of 4 input LUTs	64	49152	1%
Number of occupied slices	32	24576	1%
Number of Slices containing only related logic	32	32	100%
Number of Slices containing unrelated logic	0	32	0%
Total Number 4 input LUTs	64	49152	1%
Number of bonded IOBs	133	640	20%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	904		
Additional JTAG gate count for IOBs	6384		
Peak memory usage	259 MB		

Tabla D21: Síntesis DistanceJIandHI.vhd Se instancia 10 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	214	24576	0%
Number of 4 input LUTs	422	49152	0%
Number of bonded IOBs	535	640	83%
Timing Summary			
Minimum period / Maximum Frequency	No path found		
Minimum input arrival time before clock	No path found		
Maximum output required time after clock	No path found		
Maximum combinational path delay	13,717 ns		

Tabla D22: Map DistanceJIandHI.vhd Se instancia 10 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of 4 input LUTs	294	49152	1%
Number of occupied slices	213	24576	1%
Number of Slices containing only related logic	213	213	100%
Number of Slices containing unrelated logic	0	213	0%
Total Number 4 input LUTs	294	49152	1%
Number of bonded IOBs	535	640	83%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	2847		
Additional JTAG gate count for IOBs	25680		
Peak memory usage	265 MB		

Tabla D23: Síntesis DistanceLastStateBitIN.vhd Se instancia 10 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	44	24576	0%
Number of 4 input LUTs	82	49152	0%
Number of bonded IOBs	105	640	16%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	No path found		
Minimum input arrival time before clock	7,471 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D24: Map DistanceLastStateBitIN.vhd Se instancia 10 veces.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of 4 input LUTs	82	49152	1%
Number of occupied slices	44	24576	1%
Number of Slices containing only related logic	44	44	100%
Number of Slices containing unrelated logic	0	44	0%
Total Number 4 input LUTs	82	49152	1%
Number of bonded IOBs	105	640	16%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	795		
Additional JTAG gate count for IOBs	5040		
Peak memory usage	259 MB		

Tabla D25: Síntesis normalice.vhd			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	1274	24576	5%
Number of Slice Flip Flops	271	49152	0%
Number of 4 input LUTs	2351	49152	4%
Number of bonded IOBs	460	640	71%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period / Maximum Frequency	7,211 ns		138,682 MHz
Minimum input arrival time before clock	8,198 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D26: Map normalice.vhd			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	263	49152	1%
Number of 4 input LUTs	2291	49152	4%
Number of occupied slices	1375	24576	5%
Number of Slices containing only related logic	1375	1375	100%
Number of Slices containing unrelated logic	0	1375	0%
Total Number 4 input LUTs	2323	49152	4%
Number used as logic	2291		
Number used as route-thru	32		
Number of bonded IOBs	460	640	71%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	20930		
Additional JTAG gate count for IOBs	22080		
Peak memory usage	278 MB		

D4 síntesis SimulacionCompleta.vhd.

Está constituido por los bloques que indicamos a continuación. Todos ellos se instancian una única vez en el bloque principal SimulacionCompleta.vhd.

- SimulacionCompleta.vhd
  - ViterbiDecoder.vhd
  - EncoderK7.vhd
  - MantienePulso.vhd
  - Retardador.vhd
  - EscrituraFicheros.vhd
  - constantes.vhd

En el *apartado 7.3, tablas 7.5 y 7.6,* mostramos la información relativa a la síntesis de SimulacionCompleta.vhd. De manera que en este *apartado D.4* recopilamos esa información, añadiéndole los parámetros de síntesis de sus módulos internos.

Los datos de ViterbiDecoder.vhd están en las *tablas D3 y D4*. Y los de EncoderK7.vhd en las *tablas D1 y D2*.

EscrituraFicheros.vhd no es sintetizable, este paquete de código sólo se utiliza para simulación.

Tanto MantienePulso.vhd como Retardador.vhd los hemos diseñado de manera que sean independientes de la profundidad de memoria.

Tabla D27: Síntesis SimulacionCompleta.vhd, profundidad de memoria 36.				
Design Summary	SimulacionCompleta			ViterbiDecoder
Logic Utilization	Used	Available	Utilization	
Number of Slices	7223	24576	29%	7060
Number of Slice Flip Flops	3900	49152	7%	3712
Number of 4 input LUTs	13233	49152	25%	12488
Number of bonded IOBs	16	640	2%	8
Number of FIFO16/RAMB16s	32	320	10%	32
Number used as RAMB16s	32			32
Number of GCLKs	1	32	3%	1
Timing Summary				
Minimum period, T_{CLK} / Maximum F_{CLK}	10,204 ns	97,997 MHz		100,884 MHz
Velocidad máxima de simulación.	12,25 Mega bits/s			12,615 Mega símbolos/s
Minimum input arrival time before clock	9,6751 ns			8,852 ns
Maximum output required time after clock	5,845 ns			4,851 ns
Maximum combinational path delay	6,399 ns			No path found

Tabla D28: Map SimulacionCompleta.vhd, profundidad de memoria 36.				
Design Summary	SimulacionCompleta			ViterbiDecoder
Logic Utilization	Used	Available	Utilization	
Number of Slice Flip Flops	3897	49152	7%	3707
Number of 4 input LUTs	12733	49152	25%	12047
Number of occupied slices	8933	24576	36%	8383
Number of Slices containing only related Logic	8933	8933	100%	8383
Number of Slices containing unrelated logic	0	8933	0%	0
Total Number 4 input LUTs	12817	49152	26%	12098
Number used as logic	12733			12047
Number used as rote-thru	84			51
Number of bonded IOBs	16	640	2%	8
Number of BUFG/BUFGCTRLs	1	32	3%	1
Number used as BUFGs	1			1
Number used as BUFGCTRLs	0			0
Number of FIFO16/RAMB16s	32	320	10%	32
Number used as FIFO16s	0			0
Number used as RAMB16s	32			32
Total equivalent gate count for design	129374			122714

Tabla D29: Síntesis MantienePulso.vhd, cualquier profundidad de memoria.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	48	24576	0%
Number of Slice Flip Flops	35	49152	0%
Number of 4 input LUTs	86	49152	0%
Number of bonded IOBs	9	640	1%
IOB Flip Flops	3		
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period, T_{CLK}/ Maximum Frequency, F_{CLK}	5,235 ns		191,024 MHz
Minimum input arrival time before clock	3,875 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D30: Map MantienePulso.vhd, cualquier profundidad de memoria.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	34	49152	1%
Number of 4 input LUTs	57	49152	1%
Number of occupied slices	46	24576	1%
Number of Slices containing only related logic	46	46	100%
Number of Slices containing unrelated logic	0	46	0%
Total Number 4 input LUTs	74	49152	1%
Number used as logic	57		
Number used as route-thru	17		
Number of bonded IOBs	9	640	1%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	886		
Additional JTAG gate count for IOBs	432		
Peak memory usage	261 MB		

Tabla D31: Síntesis Retardador.vhd, cualquier profundidad de memoria.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slices	341	24576	0%
Number of Slice Flip Flops	263	49152	0%
Number of 4 input LUTs	550	49152	0%
Number of bonded IOBs	9	640	1%
Number of GCLKs	1	32	3%
Timing Summary			
Minimum period, T_{CLK}/ Maximum Frequency, F_{CLK}	9,961 ns		103,514 MHz
Minimum input arrival time before clock	9,267 ns		
Maximum output required time after clock	4,851 ns		
Maximum combinational path delay	No path found		

Tabla D32: Map Retardador.vhd, cualquier profundidad de memoria.			
Design Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	260	49152	1%
Number of 4 input LUTs	520	49152	1%
Number of occupied slices	411	24576	1%
Number of Slices containing only related logic	411	411	100%
Number of Slices containing unrelated logic	0	411	0%
Total Number 4 input LUTs	536	49152	1%
Number used as logic	520		
Number used as route-thru	16		
Number of bonded IOBs	9	640	1%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number used as BUFGCTRLs	0		
Total equivalent gate count for design	5680		
Additional JTAG gate count for IOBs	432		
Peak memory usage	263 MB		

D5 Síntesis SimulacionCompletaVerilog.vhd.

Está constituido por los bloques que indicamos a continuación. Todos ellos se instancian una única vez en el bloque principal SimulacionCompletaVerilog.vhd.

- SimulacionCompletaVerilog.vhd
  - decoderverilog.v
  - EncoderK7.vhd
  - MantienePulso.vhd
  - Retardador.vhd
  - EscrituraFicheros.vhd
  - constantes.vhd

Los datos de decoderverilog.v están en el *apartado 7.4 y la tabla 7.3*. A continuación recopilamos esta información para juntarla con la de SimulacionCompletaVerilog.vhd.

Los resultados tras síntesis y map de EncoderK7.vhd están en las *tablas D1* y *D2*.

Los resultados de MantienePulso y Retardador están en las *tablas D29* a *D32*.

EscrituraFicheros.vhd no es sintetizable, este paquete de código sólo se utiliza para simulación.

Tabla D33: Síntesis SimulacionCompletaVerilog.vhd, profundidad de memoria 32.				
Design Summary	SimulacionCompletaVerilog			Decoderverilog
Logic Utilization	Used	Available	Utilization	
Number of Slices	2008	24576	8%	1671
Number of Slice Flip Flops	1142	49152	2%	909
Number of 4 input LUTs	2907	49152	5%	2275
Number of bonded IOBs	16	640	2%	8
Number of GCLKs	1	32	3%	1
Timing Summary				
Minimum period, T_{CLK} / Maximum F_{CLK}	10,104 ns	98,975 MHz		114,378 MHz
Velocidad máxima de simulación.	12,371 Mega bits/s			14,297 Mega símbolos/s
Minimum input arrival time before clock	9,859 ns			6,850 ns
Maximum output required time after clock	5,845 ns			4,851 ns
Maximum combinational path delay	6,399 ns			No path found

Tabla D34: Map SimulacionCompletaVerilog.vhd, profundidad de memoria 32.				
Design Summary	SimulacionCompletaVerilog			decoderverilog
Logic Utilization	Used	Available	Utilization	
Number of Slice Flip Flops	1139	49152	2%	904
Number of 4 input LUTs	1824	49152	3%	1251
Number of occupied slices	2089	24576	8%	1692
Number of Slices containing only related Logic	2089	2089	100%	1692
Number of Slices containing unrelated logic	0	2089	0%	0
Total Number 4 input LUTs	2881	49152	5%	2275
Number used as logic	1824			1251
Number used as rote-thru	33			
Number used for Dual Port Rams (Two LUTs used per Dual port RAM)	1024			1024
Number of bonded IOBs	16	640	2%	8
Number of BUFG/BUFGCTRLs	1	32	3%	1
Number used as BUFGs	1			1
Number used as BUFGCTRLs	0			0
Total equivalent gate count for design	89146			83143
Additional JTAG gate count for IOBs	768			384
Peak Memory Usage	284 MB			278

GLOSARIO.

3G	Third Generation. (Tercera generación).
3GPP	Third Generation Partnership Project. (Proyecto conjunto de tercera generación).
ACK	Acknowledgment. (Acuse de recibo).
ACLR	Asynchronous Clear.
ADSL	Asymmetric Digital Subscriber Line. (Línea de abonado digital asimétrica).
AMPS	Advanced Mobile Phone System. (Sistema telefónico móvil avanzado).
ANSI	American National Standards Institute. (Instituto nacional estadounidense de estándares).
ARQ	Automatic Repeat Request. (Petición automática de retransmisión).
ASIC	Application Specific Integrated Circuit. (Circuito integrado para aplicaciones específicas).
AWGN	Additive White Gaussian Noise. (Ruido blanco Gaussiano).
BER	Bit Error Rate. (Tasa de bits erróneos).
BPSK	Binary Phase Shift Keying. (Modulación binaria por salto de fase).
BSC	Binary Symmetric Channel. (Canal binario simétrico).
CE	Clock Enable. (Habilitación de la señal de reloj).
CDMA	Code Division Multiple Acess. (Acceso múltiple por división en el código).
CLB	Configurable Logic Block. (Bloque lógico configurable).
CLK	Clock. (Reloj de un sistema digital).
COIT	Colegio Oficial de Ingenieros de Telecomunicación.
CPLD	Complex Programmable Logic Device. (Dispositivo lógico programable complejo).
DBS	Direct Broadcast Satellite. (Transmisión directa desde satélite).
DVB	Digital Video Broadcasting. (Transmisión de vídeo digital).
EEPROM	Electrically Erasable Programmable Read-Only Memory. (Memoria ROM programable y borrable electrónicamente).
EPROM	Erasable Programmable Read-Only Memory. (Memoria programable de sólo lectura y que se puede borrar).

FEC	Forward Error Correction. (Corrección de errores en el destino, a posteriori, o hacia adelante).
FIFO	First In First Out. (Primero en entrar, primero en salir).
FPGA	Field Programmable Gate Array. (Matriz de puertas programables).
GCLK	Global Clock. (Reloj global).
GSM	Global System for Mobile Communications. (Sistema global para comunicaciones móviles).
HiperLAN	High Performance Radio Local Area Network. (Protocolo de transmisión vía radio en redes de área local, normalmente menos de 1 Km de radio).
HiperMAN	High Performance Radio Metropolitan Area Network. (Protocolo de transmisión vía radio en redes de área metropolitana, decenas de kilómetros de radio).
HDL	Hardware Description Language. (Lenguaje de descripción hardware).
HDSL2	High bit rate Digital Subscriber Line 2. (Línea de abonado digital de alta velocidad binaria 2).
I^2C	Inter-Integrated Circuit. (Es un protocolo de un bus de comunicaciones serie).
IEEE	Institute of Electrical and Electronic Engineers. (Instituto de electrónica e ingenieros electrónicos).
IESS	Intelsat Earth Station Standards. (Estándares para las estaciones base terrestres en la red Intelsat).
Intelsat	International Telecommunications Satellite Organization. (Es una red de satélites de comunicaciones geostacionarios).
IOB	Input-Output Block. (Bloque de entrada-salida).
IP	Intellectual Property. (Propiedad intelectual).
IS-54	Interim Standard-54. (Estándar interno 54. Es un estándar de telefonía móvil de segunda generación, conocido como Digital AMPS(D-AMPS).
IS-95	Interim Standard-95. (Estándar interno 95. Es un estándar de telefonía móvil de segunda generación, basado en tecnología CDMA).
JTAG	Joint Test Action Group. (Es el nombre común utilizado para la norma IEEE 1149.1, titulada: Standard Test Access Port and Boundary-Scan Architecture).
LOS	Line of Sight. (Visión directa).
LTE	Long Term Evolution. (Es una evolución de la norma 3GPP).

LUT	Look-Up Table.
NACK	Negative Acknowledgment. (Acuse de recibo negativo).
NLOS	Non Line of Sight. (Sin visión directa).
OFDM	Orthogonal Frequency Division Multiplexing. (Multiplexado por división en frecuencias ortogonales).
PLD	Programmable Logic Device. (Dispositivo lógico programable).
PROM	Programmable Read-Only Memory. (Memoria programable de sólo lectura).
QPSK	Quadrature Phase Shift Keying. (Modulación en cuadratura por salto de fase).
RAM	Random access memory. (Memoria de acceso aleatorio).
S.B.	Bachelor of Science. (Título de licenciado en ciencias).
SER	Symbol Error Rate. (Tasa de símbolos erróneos).
SHDSL	Single-pair High-speed Digital Subscriber Line. (Línea digital de abonado de un sólo par de alta velocidad).
S.M.	Master of Science. (Máster en ciencias).
SPI	Serial Peripheral Interface. (Bus de interfaz de periféricos serie).
UUT	Unit Under test. (Equipo a testear).
VHDL	VHSIC HDL, Very High Speed Integrated Circuit Hardware Description Languaje. (Lenguaje de descripción hardware para circuitos integrados de muy alta velocidad).
VLSI	Very Large Scale Integration. (Integración a gran escala).
VSAT	Very Small Aperture Terminal. (Red de datos de comunicaciones por satélite que emplea una antena VSAT, terminal de apertura muy pequeña).
Wi-Fi	Wireless Fidelity
WiMAX	Worldwide Interoperability for Microwave Access. (Interoperabilidad mundial para acceso por microondas).
WLAN	Wireless Local Area Network. (Red de área local inalámbrica).
XDSL	Digital Subscriber Line Technologies. (Conjunto de tecnologías DSL)